The Amulet V2.0 Reference Manual

Brad A. Myers,
Alan Ferrency, Rich McDaniel, Robert C. Miller,
Patrick Doane, Andy Mickish, Alex Klimovitski

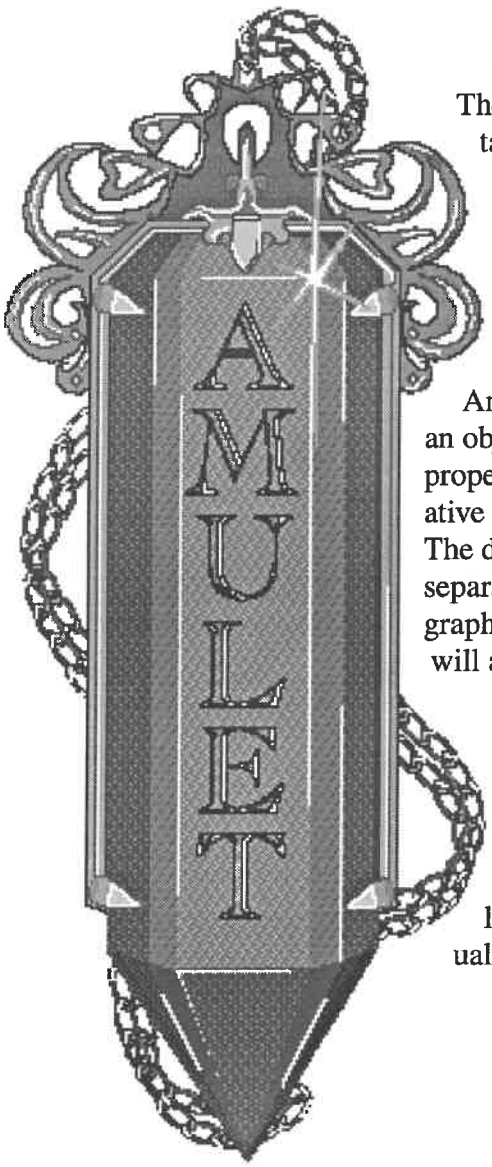
February, 1996
CMU-CS-95-166-R1
CMU-HCII-95-102-R1

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This manual describes Version 2.0 of the Amulet User Interface Toolkit, and replaces all previous versions: CMU-CS-95-166/ CMU-HCII-95-102 (June, 1995), and all change documents.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

Keywords: User Interface Development Environments, User Interface Management Systems, Constraints, Prototype-Instance Object System, Widgets, Object-Oriented Programming, Direct Maniuplation, Input/Output, Amulet, Garnet.

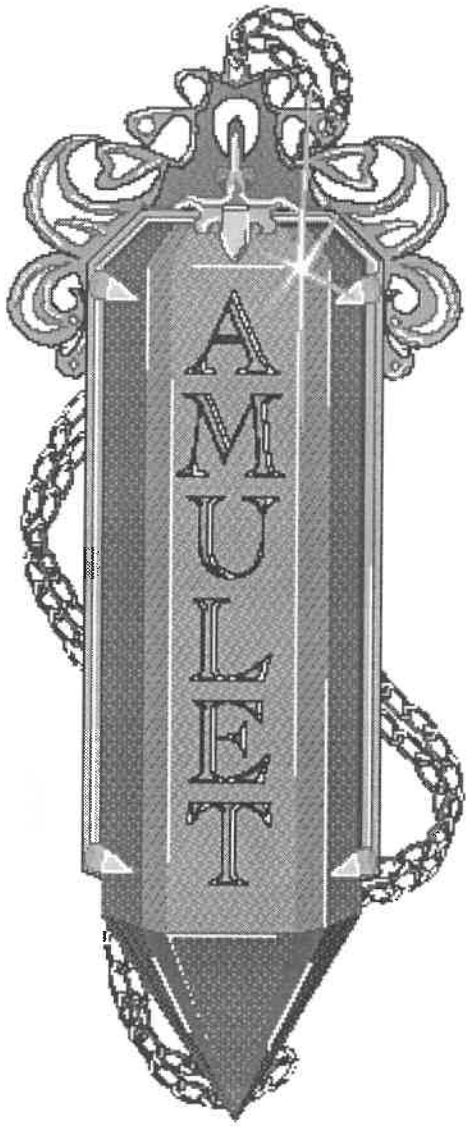


Abstract

The Amulet User Interface Development Environment contains a comprehensive set of tools that make it significantly easier to design and implement highly interactive, graphical, direct manipulation user interfaces. Applications implemented in Amulet will run without modification on Unix, PC, and Macintosh platforms. Amulet provides a high level of support, while still being Look-and-Feel independent and providing applications with tremendous flexibility. Amulet currently provides a low-level toolkit layer, which is an object-oriented, constraint-based graphical system that allows properties of graphical objects to be specified in a simple, declarative manner, and the maintained automatically by the system. The dynamic, interactive vehavior of the objects can be specified separately by attaching high-level "interactor" objects to the graphics. Higher-level tools are currently in production, which will allow user interfaces to be layed out without programming.

The Amulet toolkit is available for unlimited distribution by anonymous FTP or WWW. Amulet uses X/11 on Unix-based systems, the native Windows NT toolkit on PCs, and standard Quickdraw on the Macintosh. This document contains an overview of Amulet with download and installation instructions, a tutorial, and a full set of reference manuals for the Amulet system.

TABLE OF CONTENTS



1. Amulet V2.0 Overview

This section provides an overview of Amulet, and contains retrieval and installation instructions.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

1.1 Introduction

The Amulet research project in the School of Computer Science at Carnegie Mellon University is creating a comprehensive set of tools which make it significantly easier to create graphical, highly-interactive user interfaces. The lower levels of Amulet are called the 'Amulet Toolkit,' and these provide mechanisms that allow programmers to code user interfaces much more easily. Amulet stands for Automatic Manufacture of Usable and Learnable Editors and Toolkits.

This manual describes version 2.0 of Amulet.

Section 1.9 describes the differences between Amulet v2 and previous versions of Amulet. **Code written for Amulet v1 will need to be editied before it will compile and run properly with Amulet v2.**

The Amulet Toolkit is a portable toolkit designed for the creation of 2D direct manipulation grapical user interfaces. It is written in C++ and can be used with Unix systems running X Windows, PC's running Microsoft Windows NT or '95, or Macintosh systems running MacOS.

Amulet has been sucessfully installed on several platforms. Some of these are at CMU, and some installations are at other users' sites:

Unix:

OS: SunOS, HP/UX, Linux, IBM AIX, SGI

Compilers: gcc 2.6.3, gcc 2.7.0, ObjectCenter 2.1.0, AIX xlc, SGI CC

PC

OS: Windows NT 3.5 and 3.51 and Windows 95

Compilers: Visual C++ 2.0, 2.1 or 4.0

Mac

Compilers: Metrowerks CodeWarrior 8

We are running SunOS, HP/UX, Windows NT 3.51, and Codewarrior 8, so we'll be best able to help you if you're using one of these systems.

We have found that most users who try to use a vendor specific C++ compiler on a Unix workstation (for example, SGI's CC) run into too many language inconsistencies to successfully build Amulet. We will do what we can to help you install Amulet, but sometimes it's easier to install a new compiler (`gcc` is free) than to try to get Amulet to work on a native compiler.

There is a known bug in the `gcc` *2.5.8 compiler involving premature destruction of local variables that prevents it from being able to compile Amulet.*

Amulet provides support for color and gray-scale displays. Amulet runs on X/11 R4 through R6, using any window manager such as mwm, uwm, twm, etc. It does not use any X toolkit (such as Xtk or TCL). It has been implemented using the native Windows graphics system on the PC, and standard QuickDraw on the Macintosh. Because of stack-size limitations on the 16-bit Windows operating system, Amulet requires PC users to run Windows NT or Windows 95.

More details about Amulet are available in the Amulet home page on the World Wide Web:

```
http://www.cs.cmu.edu/~amulet
```

A previous project named Garnet was developed by the same group who is now writing Amulet. It has features similar to those in Amulet, but is implemented in Lisp. For information about Garnet, please refer to the Garnet home page:

```
http://www.cs.cmu.edu/~garnet
```

1.2 Amulet Email Addresses

There is a mailing list called `amulet-users@cs.cmu.edu` where users and developers exchange information about Amulet. Topics include user questions and software releases. To be added to the list, please send your request to `amulet-users-request@cs.cmu.edu`.

Please send questions about installing Amulet to `amulet@cs.cmu.edu`.

You can also send bug reports directly to `amulet-bugs@cs.cmu.edu`. This mail is read only by the Amulet developers.

Another mailing list, `amulet-interest@cs.cmu.edu`, is available for people who are interested in learning only about new releases of Amulet.

1.3 Using Amulet in Products: Copyright and Licensing

Amulet is available for free by anonymous FTP or WWW. Amulet has been put into the public domain. This means that anyone can use Amulet for whatever they want. In particular, Amulet can be used for commercial development without any license or fees. The resulting binaries and libraries can also be distributed commercially or for free without payments or licenses to Carnegie Mellon University (CMU). You can even include portions of the Amulet source code in your projects or products. The only restriction is that the documentation for Amulet is copyrighted, so you cannot distribute the Amulet manual or papers without permission from CMU. In return, CMU assumes no responsibility for how well Amulet works, and will not guarantee to provide any support. If you need more formal legal language, see Section 1.10 below. If you need this formally signed, then replace your company's name for COMPANY and send it back to us.

Of course, the Amulet research group would appreciate any corporate grants or donations to support the further development and maintenance of Amulet. We would also be interested in discussing grants to support adding specific features to the system that would make it more useful for your needs. Please contact Brad Myers at `bam@cs.cmu.edu` to discuss this.

If you decide to use Amulet, we would like to be able to mention this in our publicity and reports to our sponsors. (We get recognition for having users, both commercial and research projects.) Please send mail to `amulet@cs.cmu.edu` with the name of your project or product. We also like to receive screenshots. If you write papers or advertizements about systems built with Amulet, we would appreciate if you included a mention that you used Amulet, and a reference to this manual, and we would like a copy of your paper for our files.

For complete license and legal information, please see Section 1.10.

1.4 How to Retrieve and Install Amulet

You will download a different file depending on whether you're installing Amulet on a Unix, PC, or Macintosh system. You will get only the files you need to compile Amulet on your machine. If you plan to install Amulet on multiple platform types, you will need to download a different distribution for each platform type.

These instructions assume that a C++ compiler such as `gcc`, `CC`, or Visual C++ has been installed properly on your system, and you know the location of your window manager libraries, etc. *Amulet will **not** compile with* `gcc 2.5.8`.

1.4.1 The Amulet Manual

The Amulet documentation is also available for free, but it is copyrighted. This means you cannot distribute the Amulet manuals without written permission from the authors.

The Amulet manual is distributed separately from the Amulet source code. It is available in postscript format, either uncompressed, or compressed with `compress` (for UNIX) or `pkzip` (for the PC). The amulet manual is also available online. The table of contents is at:

http://www.cs.cmu.edu/afs/cs/project/amulet/amulet2/manual/Amulet_ManualTOC.doc.html

To retrieve the Amulet documentation via WWW, launch your favorite browser and go to the following URL:

```
http://www.cs.cmu.edu/~amulet/amulet2-documentation.html
```

Follow the instructions on that web page to download a copy of the Amulet manual.

To download the Amulet manual using FTP, connect to ftp.cs.cmu.edu (128.2.206.173) and login as "anonymous" with your e-mail address as the password. Type '`cd /usr0/anon/project/amulet/amulet2`' (note the double amulet's). Do not type a trailing "/" in the directory name, and do not try to change directories in multiple steps, since the intermediate directories are probably protected from anonymous access.

Set the mode of your FTP connection to binary: Some versions of FTP require you to type '`binary`' at the prompt, and others require something like '`set mode binary`'.

At the "ftp>" prompt, get the manual file you require using the get command: "ammanual.ps" is raw postscript, "ammanual.zip" is PC zip format, and "ammanual.Z" is UNIX compress format.

1.4.2 Retreiving the Amulet source code distribution

The Amulet source distribution can be retrieved via anonymous FTP, or from the Amulet WWW pages.

1.4.2.1 Retrieving the source distribution via FTP

To download the compressed Amulet source code files via FTP, you'll need an FTP client program. FTP to ftp.cs.cmu.edu (128.2.206.173) and login as 'anonymous' with your e-mail address as the password.

Type 'cd /usr0/anon/project/amulet/amulet2'. Do not type a trailing '/' in the name of the directory, and do not try to change directories in multiple steps, since the intermediate directories are probably protected from anonymous access.

Set the mode of your FTP connection to binary: Some versions of FTP require you to type 'binary' at the prompt, and others require something like 'set mode binary'.

At the "ftp>" prompt, type "get" followed by the name of the source distribution you require. The UNIX version of Amulet is called "amulet.tar.Z", the PC version is called "amulet.zip", and the Macintosh version is called "amulet.sea.hqx". For example, if you wanted the UNIX version, you'd type "get amulet.tar.Z" and press return.

1.4.2.2 Retrieving the source distribution via WWW

To download the compressed Amulet source code files from the WWW you'll need a web browser such as Netscape, Mosaic, lynx, or www. Run your browser and go to the following URL:

```
http://www.cs.cmu.edu/~amulet/amulet2-release.html
```

Scroll down to the section of the web page labelled "Downloading the current release of Amulet." Follow the link appropriate for the version of Amulet you want to download. This will automatically download the compressed Amulet source code to your machine.

1.4.3 Installing Amulet on a PC

To install Amulet on your PC, you will need to retrieve the file amulet.zip as described in Section 1.4.2.

1.4.3.1 Unpacking `amulet.zip`

Once you have retrieved the file `amulet.zip` via FTP or WWW, unzip it into the directory where you want it to reside, preserving the directory structure. For example, if you use `pkunzip` and want to have Amulet files in the `C:\AMULET` directory, at the DOS prompt type '`pkunzip -d amulet.zip C:\`'.

It is easiest to use Amulet in the base directory `\AMULET`. The tutorial and installation instructions assume you installed amulet in `C:\AMULET`. The provided `.mak` and `.mdp` files expect to find all your source files in that directory structure. If you place your Amulet directory somewhere else, you may need to change the makefiles to find your source code elsewhere. The easiest but most tedious way to do this is to use Visual C++ to remove all the old files from the project, and add all the new ones.

People familiar with PC `nmake` files may directly edit the `.mak` files to specify the location of their files. This is somewhat dangerous, because changing the wrong parts of the makefile may confuse Visual C++ and cause it to reject the file. To manually edit the makefile, use your favorite editor to search and replace all occurances of `C:\amulet` with your preferred pathname (for example, `D:\foo\bar\amulet`). If you're using Visual C++ version 4, it might be smart enough to find your files without as much work.

1.4.3.2 Windows Environment Variables

Next you should set the environment variable `AMULET_DIR` to the directory where you installed Amulet. In Windows NT, go to the Program Manager, and open the Control Panel. Choose System, and add `AMULET_DIR = C:\amulet` (substitute the appropriate pathname) to the User Environment Variables section. Amulet uses this environment variable to look for its dynamic link libraries, as well as some bitmap files in the demo programs. If the `AMULET_DIR` is not set, Amulet looks in `C:\amulet` by default.

1.4.3.3 Configuring Visual C++

In the Visual C++ menubar, choose `Tools: Options` to bring up the Visual C++ options window. Choose the `Directories` panel to access the search paths. Add the Amulet include directory `C:\AMULET\INCLUDE` to the include path, and add the Amulet library directory `C:\AMULET\LIB` to the library path. (If you installed Amulet in some other directory, be sure to specify that directory instead.)

1.4.3.4 Visual C++ Project Files

Amulet supports development with Visual C++ versions 2 and 4. These two development environments use different project files, so we've provided projects for all of the sample and demo programs for both versions of Visual C++.

Project files for use with Visual C++ version 2 have a '2' at the end of the filename (before the extension `.mak`). For example, the VC++ 2 project file that builds the Amulet library is called `amulet2.mak`. Project files for Visual C++ version 4 do not have any special characters in their names. The VC++ version 4 project files for building the Amulet library are `amulet.mak` and `amulet.mdp` (both are required to build the project). Throughout the installation instructions, reference is made to specific `.mak` files. If you use Visual C++ version 2, you should be sure to use the project file with the '2' suffix. Most of the project files are located in `C:\amulet\bin`. The sample projects are located in `C:\amulet\samples\*`. Be sure to use the correct project file for your version of Visual C++.

1.4.3.5 Compiling The Amulet Library

To build any of the sample programs or to link your own application to Amulet, you must first build the Amulet library file. The Amulet 2.0 distribution does not come with any precompiled libraries; a different version would be required for Visual C++ 2 or 4.

Launch the Visual C++ application, and open the Amulet library Visual C++ project file. This is `c:\amulet\bin\amulet2.mak` for Visual C++ V2, or `c:\amulet\bin\amulet.mdp` for Visual C++ V4. You can do this all in one step by double-clicking on the file from the File Manager.

The project files generate either `AMULETD.LIB` for a *debugging* version of the Amulet library file, or `AMULET.LIB` for a release build. These two options are called *targets* in Visual C++ v2, and *project configurations* in v4. There is an option box in the project workspace window which allows you to choose which of the versions you'd like to compile.

Choose the Build All option. Go get a cup of coffee, this'll take a while. Compiling Amulet may generate many warnings in Visual C++, but there should not be any fatal errors.

Once you have a compiled version of `AMULETD.LIB` or `AMULET.LIB`, you are ready to write your own Amulet programs and link them to the Amulet library. Section 1.4.3.6 discusses how you can build and run some of the Amulet samples and test programs. Your first experience with Amulet programming should involve the Amulet Tutorial, which includes a starter program and instructions to acquaint you with the compiling process. When you are ready to write your own Amulet program, see Section 1.4.3.8 for instructions about linking your new program to the Amulet library.

1.4.3.6 Compiling Test Programs and Demos

There are about 10 test programs included in Amulet that test the lower levels of Amulet: the ORE object system, GEM graphics routines, OPAL graphical objects, Interactors event handlers, and Widgets. The project files for these tests appear in the `amulet\bin\` directory. You can build and run these programs to test your installation of Amulet, but they are not intended to be particularly good examples of Amulet coding style. The makefiles for these programs assume you've installed Amulet in `\AMULET`, so if you want to try these programs and have installed Amulet elsewhere, you might have to change the makefiles.

There are some other example programs in AMULET\SAMPLES*. Executables are provided in the PC distribution for some of the samples. These programs are written how you should write code as an Amulet user, and are intended to be exemplary code. Each of these programs have their own .MAK file in their subdirectory, and depend on the Amulet library file AMULET.LIB. To generate an executable for any of these demos, open the project file for the program, located in its AMULET\SAMPLES* subdirectory, and build it. You can only compile the samples after you have successfully compiled the Amulet library. See Section 1.5 for descriptions of the demos, and Section 1.4.3.5 for information on compiling the Amulet library.

1.4.3.7 Using `console.cpp` to Simulate a Terminal Window

The Amulet test programs were designed in a Unix environment, and require a simulated terminal window for text output. We provide a simple way to add a console to any of your Amulet programs, using the file `console.cpp`. To include a console in your custom Amulet program, just add the file `amulet\src\gem\console.cpp` to your Visual C++ project. A console will automatically be allocated when your program starts, and destroyed when the program completes execution.

The Microsoft Windows NT console window allows selection, cutting, and pasting of text. Go to the window's menu (drag off the box in its upper left corner) to "edit," and select the submenu item "mark." This will allow you to make a selection in the window. Selecting the "edit->copy" item copies the selection to the cut buffer. If you want the default behavior to be selection mode, go to the "Settings..." menu item, and check off "Quick Edit." You can also make this the default for all future console windows by using the "Configure Default Values" menu item. In Quick Edit mode, the left button selects text, and the right button copies it to the cut buffer.

The default size of an Amulet console screen buffer is 80 by 100 characters. You can use the scroll bars in the window to look at previous lines of text. By changing the values of the `buffer_size` structure in `console.cpp` and recompiling, you can change the size of the Amulet console buffer.

1.4.3.8 Writing and Compiling New Programs Using Amulet

The easiest way to create a project file for your own Amulet program is to copy the project file from one of the sample programs (`tutorial` is a good one), remove the sample code from the project, and add your application's `.cpp` files instead. At that point you should be able to successfully compile your Amulet application.

To start from scratch, you will need to set up your Visual C++ project as follows:

- Create a new Visual C++ project of type `Application`. You do **not** need to use MFC classes in your project.
- Add your program to the project.
- If you want your program to handle terminal-style I/O, add the file `amulet\src\gem\console.cpp` to your project.
- Make sure the Amulet include and library paths are set in your Visual C++ options, as discussed in Section 1.4.3.3.

- In Visual C++ 2, choose the menu item `Project: Settings` to bring up your project settings. In Visual C++ 4, choose the menu item `Build: Settings`.
  - In the C/C++ preprocessor settings, define `NEED_BOOL` and `SHORT_NAMES`. Also define `_WINDOWS` if it is not already defined.
  - In the linker input settings, add the `AMULETD.LIB` library if you want to use the 'debugging' version of the Amulet library, or `AMULET.LIB` for a streamlined version.

Now you are ready to build your project.

1.4.3.9 PC filenames

To support as many users as possible, we are using 8 character file names with 3 character extensions for the Windows Amulet files. This manual generally refers to Unix or machine independant file names, which are often longer than 8 characters. The PC files are kept in the same directories as their equivalent Unix or Mac files. In cases where the Unix filenames are too long for the PC, the filename is logically shortened, either by truncation or removal of vowels. On the PC, the extension .cpp is used for all C++ code files, and headers use the extension .h.

1.4.4 Installating Amulet on a Unix Workstation

To install Amulet on your Unix workstation, you will need to retrieve the file `amulet.tar.Z` as described in Section 1.4.2.

1.4.4.1 Unpacking `amulet.tar.Z`

Expanding `amulet.tar.Z` into a usable Amulet directory structure is a two part process. At the Unix prompt, first type "`uncompress amulet.tar.Z`". Your copy of `amulet.tar.Z` will be replaced with `amulet.tar`. Now type "`tar -xvf amulet.tar`" at the Unix prompt to generate the `amulet/` directory tree as a subdirectory of your current directory.

1.4.4.2 Setting your Environment Variables

The Amulet Makefiles have been written so that all Amulet users must set two environment variables in their Unix shell before they can compile any program. Consistent binding of these variables by all Amulet users ensures that the installed Amulet binaries will always be compatible with user programs. Once Amulet has been installed and programs are being written that only depend on its library file, it would be possible for users to write their own Makefiles without regard to these variables. However, we recommend that all Amulet users at a site continue to use consistent values of environment variables to facilitate upgrading to future versions of the system. Typically, these environment variables will be set in your `.login` file:

- `AMULET_DIR` Set this to the root directory of the Amulet software hierarchy on your machine. The `csh` command to do this is:

```
setenv AMULET_DIR /usr/johndoe/amulet
```

`AMULET_VARS_FILE` Set this to the `Makefile.vars.*` file appropriate for your compiler and machine. Use only the filename, not the complete pathname to the file. This file will be included by the main Amulet `Makefile`.

If you are using SunOS or HP/UX, then set AMULET_VARS_FILE to one of the following files, which have been tested by the Amulet group on our own machines.

`Makefile.vars.gcc.Sun`	For `gcc` on SunOS 4.x
`Makefile.vars.gcc.HP`	For `gcc` on HP/UX 9.x
`Makefile.vars.CC.Sun`	For ObjectCenter's `CC` on SunOS 4.x
`Makefile.vars.CC.HP`	For ObjectCenter's `CC` on HP/UX 9.x

For example, if you run csh, your `.login` might contain the lines

```
setenv AMULET_DIR        /usr/johndoe/work/amulet
setenv AMULET_VARS_FILE  Makefile.vars.CC.HP
```

If you are using a different platform, you may find a file tailored to your platform in the `contrib` directory. Files in `contrib` are provided by other Amulet users. Among them you will find:

`Makefile.vars.gcc.Solaris`	For `gcc` on Solaris
`Makefile.vars.gcc.linux`	For `gcc` on linux (static library only)
`Makefile.vars.gcc.linux-ELF`	For `gcc` on linux with ELF shared libraries
`Makefile.vars.gcc.AIX`	For `gcc` on AIX
`Makefile.vars.CC.AIX`	For `xlc` on AIX

To use one of these files, first copy it to your `bin` directory, so that the Makefiles can find it, then set your AMULET_VARS_FILE variable

If you run csh, your `.login` might include the lines:

```
setenv AMULET_DIR        /usr/janedoe/amulet
setenv AMULET_VARS_FILE  Makefile.vars.gcc.linux-ELF
```

If you can't find your platform in either the `bin` or the `contrib` directory, then you should set your `AMULET_VARS_FILE` variable to the file:

`Makefile.vars.custom`	For any other configuration

If you run csh, your `.login` might include the lines:

```
setenv AMULET_DIR        /usr/jonwoo/amulet
setenv AMULET_VARS_FILE  Makefile.vars.custom
```

If you use `Makefile.vars.custom`, try compiling Amulet first. If the compilation does not finish smoothly, you probably need to make changes to the variables in `Makefile.vars.custom`. See Section 1.4.4.6 for more information. Only edit `amulet/bin/Makefile.vars.custom` while installing Amulet.

1.4.4.3 Generating the Amulet Library File

After you have set your environment variables, `cd` into the `bin/` directory and invoke `make`, with no arguments. This generates many object files, and eventually the Amulet library will be deposited in the `amulet/lib/` directory. If you are using `Makefile.vars.gcc.Sun`, `Makefile.vars.gcc.HP`, `Makefile.vars.gcc.Solaris`, or `Makefile.vars.gcc.linux-ELF`, then Amulet is compiled into a shared library called `libamulet.*` (The suffix indicating a shared library is platform-dependent, but it is typically `.so` or `.sl`.) If you are using any other compiler or platform, then Amulet is compiled into a static library called `libamulet.a`.

It is common to get many warning messages during the build, but you should have no fatal errors. If the compile procedure is interrupted by an error, you may need to customize the Makefile variables for your platform. Set your `AMULET_VARS_FILE` environment variable to `Makefile.vars.custom`, and refer to Section 1.4.4.6. Change appropriate variables in `Makefile.vars.custom`, and recompile. If you are unable to compile Amulet after trying different combinations of compiler switches, please send mail to `amulet-bugs@cs.cmu.edu` and we will try to make the Amulet code more portable.

Once you have generated the library, you are ready to write your own Amulet programs and link them to the Amulet library. Your first experience with Amulet programming should involve the Amulet Tutorial, which includes a starter program and instructions to acquaint you with the compiling process. When you are ready to write your own Amulet program, see Section 1.4.4.5 for instructions on linking your new program to the Amulet library.

1.4.4.4 Compiling Test Programs and Examples

From the `bin/` directory, doing '`make all`' generates about 10 executable binaries that test the lower levels of Amulet: ORE object system, GEM graphics routines, OPAL graphical objects, Interactors event handlers, and Widgets. You can run these programs directly, such as '`./testgem`'. You can build and run these programs to test your installation of Amulet, but they are not intended to be particularly good examples of Amulet coding style.

There are also some example programs in `amulet/samples/*`. These programs are more like you would write as an actual Amulet user, and are intended to be exemplary code. Each of these programs have their own Makefile in their subdirectory, and depend on the Amulet library file `libamulet.a` (see above). To generate binaries for these files, `cd` into their subdirectory and invoke `make` with no parameters. See Section 1.5 for descriptions of these demos.

1.4.4.5 Writing and Compiling New Programs Using Amulet

It is important to set your `AMULET_DIR` and `AMULET_VARS_FILE` environment variables, and to retain the structure of the sample Makefiles in your local version. By keeping the line '`include $(AMULET_DIR)/bin/Makefile.vars`' at the top of your Makefile, and continuing to reference the Amulet Makefile variables such as `AM_CFLAGS` and `CC`, you will be assured of generating binary files compatible with the Amulet libraries.

When you are ready to write a new program using Amulet, it is easiest to start with an example Makefile. The easiest way to start a new project is to copy the contents of `samples/tutorial/` into a new directory, and edit the `Makefile` and `tutorial.cc` files to begin your project. Invoking `make` in that directory generates a binary for your Amulet program. The examples presented in the Amulet Tutorial all start from this point, and can be used as models for your programs.

1.4.4.6 Customizing the Makefile.vars.custom Variables

C++ is a standardized language, but many compilers do not fully support the ANSI standard completely or correctly. Because of this, Amulet requires different source code in certain places with various platform/ compiler combinations. We handle this with conditional code that depends on the compile-time definition of the variables defined in your `AMULET_VARS_FILE`. These variables need to be set apropriately for your system. This section is a guide to the variables that control the conditional Amulet code. If, after experimenting with the compiler variables documented below, you are still not able to successfully compile Amulet, please send mail to `amulet-bugs@cs.cmu.edu` so we can try to make the Amulet code more portable.

Amulet will not compile in `gcc` 2.5.8, *due to a compiler bug.*

Before changing any of the Makefile variables, you should try compiling Amulet once with one of the default `Makefile.vars.*` files. If the procedure does not terminate smoothly, you should have some indication of what switches need to be added or changed. Make sure that your `AMULET_VARS_FILE` environment variable is set to `Makefile.vars.custom` (found in `amulet/bin`), and bring this file up in an editor. This is the only file that you should change.

The variables that control conditional Amulet code are defined with `-D` compiler switches. For example, we have found that the `CC` libraries on Suns do not provide the standard function `memmove()`, so we have to define it ourselves in Amulet. The Amulet version of `memmove()` is only defined when the compiler switch `-DNEED_MEMMOVE` is included in the compile call (which declares the variable `NEED_MEMMOVE`). By adding or removing the `-DNEED_MEMMOVE` switch, you control whether Amulet defines `memmove()`.

The interface for defining these variables is the `AM_CFLAGS` list in `Makefile.vars.custom`. For each variable `VAR`, you would include the switch `-DVAR` in the `AM_CFLAGS` list to define the variable, or simply leave out the switch to avoid defining the variable. By iteratively adding or removing these variables from your `AM_CFLAGS` list and recompiling Amulet, you should be able to install Amulet on your system.

1.4.4.6.1 Compiler Variables

`HP` Including `-DHP` in the `AM_CFLAGS` list will cause some type casting required for HP's that is inappropriate on other machines.

`GCC` Including `-DGCC` in the `AM_CFLAGS` list causes different header files to be referenced than when Amulet is compiled with `CC` or Visual C++.

`NEED_BOOL` Including `-DNEED_BOOL` in the `AM_CFLAGS` list causes Amulet to define

the `bool` type. This type is pre-defined in `gcc`.

`NEED_MEMMOVE` Including `-DNEED_MEMMOVE` in the `AM_CFLAGS` list causes Amulet to define its own `memmove()` function. This function is missing in some C libraries.

`NEED_STRING` Including `-DNEED_STRING` in the `AM_CFLAGS` list causes Amulet to include the standard header file `strings.h` in special places required by some versions of the `CC` compiler.

`DEBUG` Including `-DDEBUG` in the `AM_OP` list causes Amulet debugging code to be compiled into your binaries. If you do not define `DEBUG`, you will lose a lot of runtime debugging information, such as slot, object, and wrapper names, and tracing and breaking in the debugger, but your executable will be smaller and it will run faster. This switch is *not* the same as the `-g` option, which compiles debugging symbols for a debugger such as `gdb` into your executable. Using the `DEBUG` switch compiles in Amulet-specific runtime debugging information which makes it easier to debug your Amulet application while it's running. The options `-g` and `-DDEBUG` are mutually exclusive; neither is required to use the features the other provides.

1.4.4.6.2 Makefile Variables

`CC` Your compiler, such as `/usr/local/bin/gcc`.
If you're compiling using ObjectCenter's CC, you may have to specify the pathname explicitly. Otherwise, a default (non-Centerline) CC may be used which will not successfully compile Amulet.

`LD` Your linker, such as `/bin/ld`. This command is used to link libraries into archive files.

`AM_OP` Options you want to pass to your compiler, such as `-g` to put `gcc` debugging information in the binaries, `-O` to enable optimization, or `-pg` to enable profiling.

`AM_CFLAGS` A list of switches to pass to the compiler, including the Amulet compiler variables listed above and any include and library paths you need to specify.

`AM_LDFLAGS` A list of switches to pass to the linker.

`AM_LIBS` A list of libraries to link with your program, such as `-lX11` and `-lg++`.

`AM_LIB_SHARING` A symbol indicating which kind of library should be built: either `static` to make a static Amulet library (`libamulet.a`), or `shared` to make a shared library. This option will build shared libraries only if your compiler is `gcc`, version 2.7.0 or newer.

`AM_SHLIB_SUFFIX` The appropriate suffix for a shared library on your platform. On HP/UX, for example, shared libraries end with "`.sl`", but on Solaris and linux, shared libraries end with "`.so`".

1.4.4.6.3 Command Line `make` options

`Make` allows you to specify override values for makefile variables on the command line. This allows quick testing of various Makefile configurations without having to edit the Makefile.vars.* file every time you want to try something new. To override a makefile variable on the command line, just type `make <variable>=<value> <target>`. For example, to try compiling testselectionwidget with shared libraries turned off, type:

```
make AM_LIB_SHARING=static testselectionwidget
```

1.4.4.6.4 Xlib Pathnames

Some compilers on Unix systems do not know where to find the Xlib library and include files. You may need to include the pathnames for your Xlib files in the `AM_CFLAGS` list in `Makefile.vars.custom`. The include path should be provided with the `-I` switch, and the library path should be provided with the `-L` switch.

For example, some HP machines store their Xlib include and library files in `/usr/include/X11R5/` and `/usr/lib/X11R5/`. If this is how your machine is set up, your `AM_CFLAGS` definition would look like this:

```
AM_CFLAGS =-$(AM_OP) I$(AMULET_DIR)/include -DGCC -DDEBUG -DHP \
        -I/usr/include/X11R5 -L/usr/lib/X11R5
```

Note that a backslash is required at the end of a line when the definition of a Makefile variable continues on the next line.

If you are having trouble finding your Xlib include and library files, try looking in these directories:

```
/usr/include/X11R?, /usr/lib/X11R?
/usr/local/X11R?/include, /usr/local/X11R?/lib
```

1.4.4.7 Why is my application so big?

Many users complain that applications that link to the Amulet library are very large. It is not unusual to have a hello world program compile down to a few megabytes of executable. Obviously, not all of this is useful, necessary information. There are several things you can do to reduce the size of your executables.

- Recompile the Amulet library and your application without the -DDEBUG -g switches to produce your final release. This removes symbolic debugging information, so you can't debug the resulting executable with a debugger such as `gdb`. It also disables a lot of code (such as the Inspector) that won't be needed in applications that you plan to ship to a customer.
- Compile Amulet into a shared library, if your platform allows it. Amulet is automatically compiled to shared libraries if you use `gcc 2.7.0` or higher on SunOS, HP/UX, Solaris, or linux. It may also be possible on other compilers; consult your compiler's documentation.

Using a shared library will make all of your Amulet applications smaller, but you must distribute all the shared libraries with your applications and install the libraries where they can be found at run-time.

- Run `strip` on your executables. This is one of the most effective way of shrinking your executables. Just type "`strip`" followed by the name of the program that's to large. `Strip` takes out all of the symbol tables, which are necessary at compile and link time, but are wasted space at run time. If you are using shared libraries, you may also want to also `strip` the Amulet shared libraries before distributing them with your application.

1.4.5 Installing Amulet on a Macintosh

To install Amulet on your Macintosh, you will need to retrieve the file `amulet.sea.hqx` as described in Section 1.4.2. The Macintosh version of Amulet is being developed with Metrowerks CodeWarrior 8. The code will run with CodeWarrior 7 as well, but you will need to make your own new project files (see Section 1.4.5.6) for the tests and demos. The project files we provide will work on CodeWarrior 8, but not on CodeWarrior 7 without modification.

1.4.5.1 Unpacking `amulet.sea.hqx`

Once you have retrieved the file `amulet.sea.hqx` via FTP or WWW, unbinhex it. Some communications programs (such as Netscape and Fetch) do this for you automatically. If yours does not, try using Stuffit Expander. Run the application `amulet.sea` and select the directory where want the Amulet source tree to reside.

1.4.5.2 Macintosh Environment Variables

Amulet needs to know the location of your amulet directory for some bitmap files in the demo programs. Use SimpleText to create an ASCII text file called `amulet.env` in the Preferences folder in the System Folder of your boot drive. This text file must contain the full pathname to your amulet directory. For example, if your boot drive is called `Hard Disk` and the amulet folder is called `amulet` then the pathname would be `Hard Disk:amulet`. Note that there should be no colon at the end of the pathname and that this file must be an ASCII text file.

1.4.5.3 Creating the Precompiled Header file

Launch the CodeWarrior IDE application and open the CodeWarrior project file named `AmuletHeaders68K.proj` if your Macintosh is running a 680x0 processor or open `AmuletHeadersPPC.proj` if your Macintosh is running a PowerPC processor. These project files are located in `amulet:MAC:pch`. Then, select the `AmuletHeaders.pch` file in the project window and choose `Compile` from the `Project` menu.

1.4.5.4 Compiling The Amulet Library

To build any of the sample programs or to link your own application to Amulet, you must first build the Amulet library file. The Amulet 2.0 distribution does not come with any precompiled libraries.

Launch the CodeWarrior application, and open the Amulet library CodeWarrior project file called `amulet:MAC:amulet68K.proj` if your Macintosh is running a 680x0 processor or open `amulet:MAC:amuletPPC.proj` if your Macintosh is running a PowerPC processor.

Choose Make from the Projct menu. Go get a cup of coffee, this'll take a while. Compiling Amulet may generate many warnings in Metrowerks, but there should not be any fatal errors. If you do get any fatal errors, please send email to `amulet-bugs@cs.cmu.edu` so we can try to help you with your installation.

Once you have a compiled version of the Amulet library you are ready to write your own Amulet programs and link them to the Amulet library. Section 1.4.5.5 discusses how you can build and run some of the Amulet samples and test programs. Your first experience with Amulet programming should involve the Amulet Tutorial, which includes a starter program and instructions to acquaint you with the compiling process. When you are ready to write your own Amulet program, see Section 1.4.5.6 for instructions about linking your new program to the Amulet library.

If you would like to build a version of the Amulet libary without debugging code, you should open the project file `amulet:MAC:amuletNoDebug68K.proj` if your Macintosh is running a 680x0 processor or open `amulet:MAC:amuletNoDebugPPC.proj` if your Macintosh is running a PowerPC processor. You should also change the Prefix file in the C/C++ Language Settings for your project file (from the Preferences Command in the Edit Menu) to be "Am_Prefix.h" instead of "Am_DebugPrefix.h".

1.4.5.5 Compiling Test Programs and Demos

There are about 10 test programs included in Amulet that test the lower levels of Amulet: the ORE object system, GEM graphics routines, OPAL graphical objects, Interactors event handlers, and Widgets. The project files for these tests appear in the `amulet:MAC` directory. You can build and run these programs to test your installation of Amulet, but they are not intended to be particularly good examples of Amulet coding style.

There are some other example programs in `AMULET:SAMPLES:*`. These programs are written how you should write code as an Amulet user, and are intended to be exemplary code. Each of these programs have their own project files located within the sample's directory, and depend on the Amulet library file. To generate an executable for any of these demos, open the project file for the program, and choose Make from the Project Menu. You can only compile the samples after you have successfully compiled the Amulet library. See Section 1.5 for descriptions of the demos, and Section 1.4.5.4 for information on compiling the Amulet library.

The Amulet demos currently have no feedback during the launch process so it may appear that the program has crashed while it is still loading.

1.4.5.6 Writing and Compiling New Programs Using Amulet

The easiest way to create a project file for your own Amulet program is to start with the stationary project files that are provided with the Amulet distribution. They are `amulet:MAC:am_stationary68K.stat` for 680x0 processors and `amulet:MAC:am_stationaryPPC.stat` for PowerPC processors. Open the appropriate project file, save it as your new project file and add your application's `.cpp` files instead. At that point you should be able to successfully compile your Amulet application.

To start from scratch, you will need to set up your CodeWarrior project as follows:

- Create a new Metrowerks project file of type MacOS C/C++
- Add your program to the project.
- Add the file `amulet.rsrc` to the project. This file contains important Amulet resources. Without it, you will get titlebars on your Amulet Menus, and other ugly effects.
- Add an access path to the amulet directory in the Access Paths section of the Project Preferences.
- The following settings should be turned on in the Language section of the Project Preferences: `Activate C++ Compiler, ANSI Keywords Only, Enums always Int. ANSI Strict` should be turned off. Set the prefix file to `Am_DebugPrefix.h` if you are using the debugging version of the Amulet library or `Am_Prefix.h` if you are not using the debugging version of the Amulet library.
- For 680x0 projects, the `Code Model` should be set to `Large` and `Far Data` should be turned on
- Set the Project Type to be application (Type 'APPL') and set both the Preferred Heap Size and Minimum Heap Size = 4096K..

Now you are ready to build your project.

1.5 Test Programs and Demos

The procedure for compiling and executing the demos and test programs is different depending on your platform. See Section 1.4.3 for PC-specific instructions, or Section 1.4.4 for Unix-specific instructions on installing Amulet and compiling the tests and samples.

The following list describes each of the sample programs provided with Amulet:

`hello` This program creates a window and displays 'hello world' in it. You can exit by hitting `meta-shift-f1`.

`goodbye_button` This program creates a button widget on the screen which quits when pressed.

`goodbye_inter` This program displays "goodbye world" in a window, and an interactor causes the program to quit when the text is clicked on.

`tutorial` This program simply creates a window on the screen. It is the starting point for all the examples in the Amulet Tutorial.

`checkers` This is a two-player checkers game that demonstrates the multi-screen capabilities of Amulet in Unix. You can run it on two screens by supplying the names of the displays when the program is executed, as in '`checkers my_machine:0.0 your_machine:0.0`'. On the PC or Macintosh, you can still run this program, but it can only be displayed on one screen.

`space` This program looks a little bit like NetTrek, and demonstrates many features of Amulet: constraints, bitmaps, polylines, widgets, and scrolling windows. Some of the interactions to try are:

- Leftdown in the background of the Short-Range Scan creates a ship
- Leftdown drag on an existing ship moves it
- Middledown+drag (`META`-leftdown+drag on PC, `OPTION-SHIFT`-leftdown+drag on the Macintosh) from one ship to another draws a phaser beam and destroys the destination ship
- Rightdown+drag (`OPTION`-leftdown+drag) from one ship to another establishes a tractor beam which stays attached to the ships through constraints
- Dragging the white rectangle in the Long-Range Scan changes the visible area in the Short-Range Scan. You can scroll the visible area with the scroll-bars or by dragging the white feedback rectangle in the Long-Range Scan.

`agate` This program is used to train a new gesture classifier for use by the gesture interactor. See Section 5.3.5.6 for more information about the gesture interactor, `agate`, and gesture classifiers.

1.6 Amulet Header Files

Usually the only Amulet header file you'll need to include in your Amulet programs is `amulet.h`, which will include all of the others for you. If you like to use header files as a reference manual, you can find them in the `include/amulet/` directory.

Amulet header files fall into two categories, Basic, and Advanced. Basic header files define all the standard objects, functions, and global variables that Amulet users might need to write various applications. Advanced header files define lower level Amulet functions and classes which would be useful to an advanced programmer interested in creating custom objects in addition to the default objects Amulet supports. Most users will only ever include the Basic header files.

1.6.1 Basic header files

The header file `amulet.h` includes all the headers most amulet programmers will ever need to use: `standard_slots.h`, `value_list.h`, `gdefs.h`, `idefs.h`, `opal.h`, `inter.h`, and `widgets.h`. Here is a summary of the basic header files, listing the major objects, classes, and functions they define.

- `gdefs.h`: `Am_Style`, `Am_Font`, `Am_Point_List`, `Am_Point_Array`, `Am_Image_Array`
- `idefs.h`: `Am_Input_Char`, predefined `Am_Input_Char`'s
- `object.h`: `Am_Object`, `Am_Slot`, `Am_Instance_Iterator`, `Am_Slot_Iterator`, `Am_Part_Iterator`

- types.h: Am_String, Am_Value, Am_Error, Am_Wrapper, Am_Method_Wrapper
- standard_slots.h: standard slot names, Am_Register_Slot_Key, Am_Register_Slot_Name, Am_Get_Slot_Name, Am_Slot_Name_Exists
- opal.h: predefined Am_Style's, predefined Am_Font's, Am_Screen, Am_Graphical_Object, Am_Window, Am_Rectangle, Am_Roundtangle, Am_Line, Am_Polygon, Am_Arc, Am_Text, Am_Bitmap, Am_Group, Am_Map, predefined formulas, Am_Initialize, Am_Cleanup, Am_Beep, Am_Move_Object, Am_To_Top, Am_To_Bottom, Am_Create_Screen, Am_Do_Events, Am_Main_Event_Loop, Am_Exit_Main_Event_Loop, predefined Am_Point_In_ functions, Am_Translate_Coordinates
- formula.h: Am_Formula, Am_Define_Formula, Am_Define_Value_Formula
- value_list.h: Am_Value_List
- text_fns.h: all text editing functions, Am_Edit_Translation_Table
- inter.h: Am_Inter_Location, default interactor method types, Am_Interactor, Am_Choice_Interactor, Am_New_Points_Interactor, Am_One_Shot_Interactor, Am_Move_Grow_Interactor, Am_Text_Edit_Interactor, Am_Where_Function's, interactor debugging functions, Am_Command, Am_Choice_Command, Am_Move_Grow_Command, Am_New_Points_Command, Am_Edit_Text_Command, Am_Undo_Handler, Am_Single_Undo_Object, Am_Multiple_Undo_Object, Am_Abort_Interactor, Am_Stop_Interactor, Am_Start_Interactor, interactor tracing, undo handlers, Am_Pop_Up_Window_And_Wait, Am_Finish_Pop_Up_Waiting
- widgets.h: Am_Border_Rectangle, Am_Button, Am_Button_Panel, Am_Checkbox_Panel, Am_Radio_Button_Panel, Am_Menu, Am_Menu_Bar, Am_Option_Button, Am_Menu_Line_Command, Am_Vertical_Scroll_Bar, Am_Horizontal_Scroll_Bar, Am_Scrolling_Group, Am_Text_Input_Widget, dialog boxes, Am_Selection_Widget, Am_Selection_Widget_Select_All_Command, Am_Text_Input_Dialog, Am_Choice_Dialog, Am_Alert_Dialog, Am_Start_Widget, Am_Abort_Widget, Am_Global_Clipboard, Am_Graphics_Set_Property_Command, Am_Graphics_Clear_Command, Am_Graphics_Clear_All_Command, Am_Graphics_Copy_Command, Am_Graphics_Cut_Command, Am_Graphics_Paste_Command, Am_Undo_Command, Am_Redo_Command, Am_Graphics_To_Bottom_Command, Am_Graphics_To_Top_Command, Am_Graphics_Duplicate_Command, Am_Graphics_Group_Command, Am_Graphics_Ungroup_Command, Am_Show_Alert_Dialog, Am_Get_Input_From_Dialog, Am_Get_Choice_From_Dialog, Am_Show_Dialog_And_Wait
- debugger.h: the inspector
- undo_dialog.h: the selective undo dialog box

1.6.2 Advanced header files

All other header files are considered Advanced header files. They support advanced Amulet features, such as user-defined objects, daemons, constraints, and so on. Most users should never include these files explicitly. For users who will be using Amulet's advanced features, here is a brief summary of the contents of each advanced header file.

- gem.h: Am_Drawonable, Am_Input_Event, Am_Input_Event_Handlers, Am_Region

- am_io.h: Am_TRACE
- object_advanced.h: Am_Demon_Queue, Am_Demon_Set, Am_Constraint, Am_Constraint_Iterator, Am_Dependancy_Iterator, Am_Slot_Advanced, Am_Object_Advanced, Am_Constraint_Context, Ore_Initialize
- priority_list.h: Am_Priority_List_Item, Am_Priority_List
- symbol_table.h: Am_Symbol_Table
- types.h: NULL, Am_Error, Am_Wrapper, Am_WRAPPER_DECL, Am_WRAPPER_IMPL, Am_WRAPPER_DATA_DECL, Am_WRAPPER_DATA_IMPL
- formula_advanced.h: Am_Formula_Advanced, Am_Depends_Iterator
- opal_advanced.h: Am_Aggregate, advanced opal object slots, Am_Draw_Method, Am_Draw, Am_Invalid_Method, Am_Invalidate, Am_Point_In_Obj_Method, Am_Translate_Coordinates_Method, Am_State_Store, Am_Item_Function, Am_Invalid_Rectangle_Intersect, Am_Window_ToDo
- inter_advanced.h: Am_Initialize_Interactors, Am_Interactor_Input_Event_Notify, Am_Inter_Tracing, Am_Get_Filtered_Input, Am_Modify_Object_Pos, Am_Choice_Command_Set_Value, interactor methods, command methods, Undo handler functions
- widgets_advanced.h: Computed_Colors_Record, Am_Checkbox, Am_Radio_Button_Item, Am_Menu_Item, widget drawing functions, widget formula constraints, other widget support functions

1.6.3 Standard Header File Macros

The actual names of the header files described above are slightly different on each machine. Long names are truncated on the PC, as described in Section 1.4.3.9. Directory paths aren't specified in the same way on the Mac as they are on Unix. To allow these files to be included simply without explicit conditional compilation on the various platforms, we've provided a set of C++ `#defines` which give each `.h` file a standard identifier on all platforms. These `#defines` are in `amulet/include/am_inc.h`.

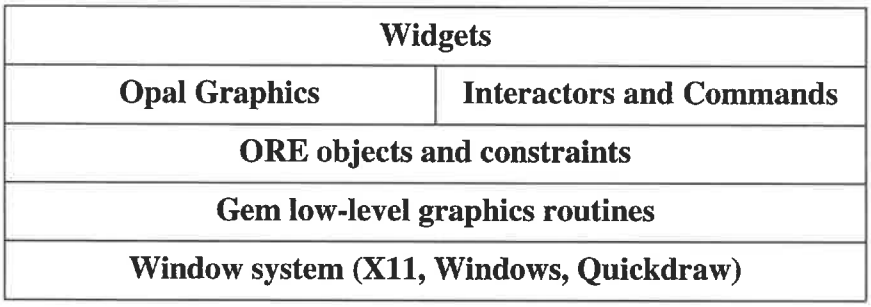
You should usually only need to `#include <amulet.h>` in your program. If you need to include any of the other header files, you should `#include <am_inc.h>` and then include the `#defined` names of the header files as found in `am_inc.h`, instead of the file's actual name. This helps ensure machine independant compilation. For example, if you needed to include `object_advanced.h` and `opal_advanced.h`, the top of your program might look like:

```
#include <amulet.h>
#include <am_inc.h>
#include OBJECT_ADVANCED__H
#include OPAL_ADVANCED__H
```

Visual C++ sometimes has problems with am_inc.h if you have the *precompiled headers* option turned on in your project. Most of the time, building the project again fixes the problem (you don't need to do a Build All). If you get errors such as, "`#include expected a filename, found an indentifier,`" this is the problem you're experiencing. To avoid it in the future, turn off precompiled headers in your project.

1.7 Parts of Amulet

Amulet is divided into several layers, which each have their own interface (and chapter of this manual). The overall picture is shown below.



The Gem layer provides a machine-independent interface so the rest of Amulet is independent of the particular window system in use. Most programmers will not need to use Gem. Ore provides a prototype-instance object system and constraint solving that is used by the rest of Amulet. Gem and ORE can each be used as a standalone system without any of the rest of Amulet, if specific functionality is required without the overhead of the higher levels of code. Opal provides an object-oriented interface to the Gem graphics system, and the Interactors and Command objects handle processing of input events from Gem. At the top is a set of Widgets, including scroll bars, buttons, menus, text input fields, and dialog boxes.

1.8 Known Bugs

We know about several bugs in Amulet which users might come across, which we have not found solutions for yet. Some of them are listed here.

1.8.1 Linux bugs

Many people experience various problems compiling and running Amulet on Linux platforms. Some people get internal compiler errors when trying to compile with `gcc`, particularly `gcc 2.7.2`. These internal compiler errors are not consistent. They happen in a different section of code each compile, and usually a `make clean` prevents any of the errors from occurring. Some people compile and link successfully, but then their executable won't run because Linux claims it's not a properly executable file. These bugs might be due to improper Linux/ gcc installations, or to bugs in compiler or OS software. We haven't found any way to fix these problems yet, and we can't reproduce them on the Linux machines we have access to, so we've been unable to find workarounds

Other people have compiled and run Amulet successfully under Linux, but come across runtime bugs, mostly dealing with the Inspector. Sometimes if you hit `^q` or choose the `Objects:Done` or `Objects:Done All` menu items, Amulet will crash. The menu items `View:Hide Internal Slots` and `View:Hide Inherited Slots`, among others, don't do anything. We've traced all of these bugs down to a single problem where gcc seems to be passing a parameter by reference instead of by value, giving us bogus pointers. We're surprised that this bug doesn't cause more widespread problems. We believe it is a problem with the Linux gcc compiler, but so far we have been unable to construct a small breaking test case, so we can't be sure of this yet.

1.8.2 Visual C++ bugs

Visual C++ 4.1 gets an internal compiler error when trying to compile the dotted/dashed line code in gwline.cpp. Because we don't have access to VC++ 4.1 to debug the problem, we've added a conditional compiler directive to compile out the inline assembly code when compiling with VC++ version 4.1 or newer. Your Amulet code shouldn't crash, but dotted and dashed lines may not be drawn correctly when you compile with VC++ 4.1.

1.8.3 Macintosh Bugs

Dotted and dashed lines aren't supported correctly on the Macintosh. For best results using dotted and dashed lines on any platform, you should use the predefined `Am_Styles` `Am_Dotted_Line` and `Am_Dashed_Line` instead of defining your own custom dashed and dotted lines. These predefined styles will be supported in future versions of Amulet, but the dashing and dotting customizations will probably not be supported.

1.9 Changes since Version 1.2

There have been *many* changes in Amulet since version 1. In particular, **all code that worked with V1.2 must be edited to compile with version 2** (for details, see Section 1.9.7). The following sections provide a summary of the changes, but you should look in the specific sections of this manual for complete information about Amulet's new features.

1.9.1 Changes from V2.0 beta to V2.0 official release

1.9.1.1 Minor Changes

- Unix/X now correctly maps some previously unmapped keysyms.
- Inspector startup keys can be reset at runtime from user code.
- Added out of memory dialog box on the Mac.
- Added minimal support for dotted and dashed lines on the Mac.
- Some options set correctly in Mac Project files.
- New 68k and PPC project files for goodbye demos.

- Mac now distinguishes between `esc` and `clear` keys correctly.

1.9.1.2 Bug Fixes

- Fixes for GIF files on the PC. Color GIF images only worked for certain bit depths on the PC in older versions of Amulet. All GIF files were occasionally corrupted in the lower right hand corner.
- Fix for fonts under Unix/X. Fonts constructed by name sometimes came up underlined when they shouldn't have.
- Bug fix on Mac to make inverted text styles work correctly.
- Bug fixes to prevent windows from shrinking and sliding around when they're not supposed to on Windows '95 and fvwm.
- Bug fix on PC for garbage-filled popup windows in menus.
- Bug fix in Windows '95 to support 16 color mode correctly.
- Patch so dotted/dashed line code doesn't cause an internal compiler error on VC++ 4.1.
- Bug fix on Mac prevents applications from crashing on exit if you close windows from the window manager.

1.9.2 Changes from V2.0 alpha to V2.0 beta

We have made a few changes between the alpha and beta releases of V2.0. They are described in this section. These should not require editing your code, if you have already upgraded to V2.0alpha.

1.9.2.1 Major Changes

- Added support for the Macintosh

1.9.2.2 Minor Changes

- New `Am_Tab_To_Next_Widget_Interactor` for handling tabbing from text field to text field in a dialog box.
- To support "weird" characters at the top of fonts, changed so META-character will set the high bit when typed. Thus META-6 gets the paragraph sign (char 182, 0xb6). Using the As_String or cout form still gets the META-6 version. Fixed Opal to pass through these characters.
- Scrolling Groups have new `Am_LINE_STYLE` slot for the outline of the scrolling area, which can be `NULL`.

1.9.2.3 Bug Fixes

- Got rid of memory leaks in testselectionwidgets.
- Makefiles for gcc with shared libraries now works

- Click in text field but not over the text makes cursor go to end instead of to the beginning
- Fixed bug where option_button popup menu didn't start off in right place.
- Fixed bug where MS Win95 windows keep moving or shrinking after the user moves or changes size.
- A fix for slots declared Am_Local, that showed up with instances of Am_Option_Buttons

1.9.3 Major Changes between V1.2 and V2.0alpha

- Support for gesture recognition, through the new Gesture_Interactor and the interactive tool Agate which allows gestures to be defined by demonstration.
- A new undo model that supports selective undo, redo and repeat of all operations. The design is described in a conference paper.
- New widgets:
  - Graphics Selection Handles widget
  - Built-in Command objects for cut, copy, paste, to top and bottom, group and ungroup, etc.
  - Option button widget, that shows one value and pops up a menu
  - Built-in dialog boxes for errors and simple queries
- Significantly expanded debugging facilities.
  In the interactive Inspector:
  - PopUp windows for constraint dependencies, slot properties and object hierarchies
  - Showing which constraints are inherited
  - Ability to trace or break when slots are set
  - Formulas and Methods now show their names
  - Many more kinds of values can be edited
  - Flashing the selected object
  - Controlling the Interactor tracing options interactively
  - Automatic update of the display when slots change

  Also, tracing and breaking on slot change; many more types print out meaningfully for cout.
- Support for GIFs (pixmaps) on Unix, PC and Mac
- Support for double buffering.

1.9.4 Minor Changes between V1.2 and V2.0alpha

- Menubars now support accelerator characters.
- For text_interactor and text_input_widgets, clicking outside now stops the interactor or widget, but passes through the click so it still does whatever other action is desired.

- Provided routines to start, stop and abort widgets and interactors: Am_Start_Widget, Am_Stop_Widget, Am_Abort_Widget; Am_Start_Interactor, Am_Stop_Interactor, Am_Abort_Interactor
- References in formulas to slots and objects that do not yet exist leave the formula uninitialized, which means that now you need fewer calls to Valid(). In particular, it is safe to do

```
obj.GV_Object(SLOT).GV(Am_LEFT)
```

even if the SLOT of obj does not yet contain a value. The formula must return a "real" value before the result of the formula is used (such as to draw the object).

- You can now destroy windows using the window manager's Kill-Window or close box, and this is converted into a message to the window. By default this deletes the window, but you can override with other behaviors.
- Added support for popping up modal dialog boxes and waiting for the user to supply a value.
- We significantly sped up the execution. Polygons got about 10 times faster, other parts are about twice as fast.
- A new type of group was added: Am_Resize_Parts_Group which acts like a regular group, except that if you change the width and height of the group, it changes the width and height of the parts correspondingly.
- The Amulet libraries can now be dynamically linked using gcc, which saves LOTS of time when compiling and linking. This is the default.
- Amulet now works on some new platforms and compilers. List of known working platforms:

Unix:

OS: SunOS, HP/UX, Linux, IBM AIX, SGI

Compilers: gcc 2.6.3, gcc-2.7.0, ObjectCenter 2.1.0

PC

OS: Windows NT 3.5 and 3.51 and Windows 95

Compilers: Visual C++ 2.0, 2.1 or 4.0

Mac

Compilers: Metrowerks CodeWarrior 7 and 8

1.9.5 Very Minor Changes between V1.2 and V2.0alpha

- Under X, extra move events are flushed to get better performance
- Guarantee that Am_SAVED_OLD_OWNER of command in inter or widget is always the inter or widget so you can use it in the DO method or constraints.
- In the PC version, we use console.cpp instead of GW Streams
- You can scroll and cut and paste in the DOS window for debugging
- obj.Text_Inspect(slot) prints out lots of information (like the inspector) which can be useful for debugging. This can be invoked from many debuggers. Also, obj.Get_Name() can be invoked from the debugger.

- New value list method: Append
- Under ObjectCenter, you no longer need to have 2-line definition of objects. Thus, it now works to do Am_Object o = obj.Get(SLOT);
- Am_Point_In_Leaf and Am_Point_In_Part (in opal.h) have extra parameter want_groups, that if false makes it not return a group. Default = true, which is the same as the v1.0 behavior.
- None of the debugging and printing code is included in the generated binaries if you compile with DEBUG off. This significantly reduces code size.
- Advanced slot properties can now be set from the object so you don't need to use object_advanced.h, including the inherit rule, read-only, single constraint mode, and demon bits.
- Built-in support for moving objects across multiple windows in the Move-Grow Interactor. The feedback object in the move-grow, new_point and gesture interactors can be a little window (so you can see it in the background), or if you use a regular object, the interactors will move the object into the appropriate window.
- Unnamed parts are now inherited.
- Interactors now will beep when aborted explicitly. You can turn this off using the Am_INTER_BEEP_ON_ABORT slot.
- Support pending delete, where the text is deleted if you type something, in the text interactor and text input widget.
- Am_Point_List class for polygons has many more features and has a consistent interface with the other list-like classes. Polygons will now be scaled correctly if their Am_WIDTH and Am_HEIGHT are set.
- Get_Object and GV_Object method in objects that declares that the return type of the slot is an object, primarily to help construct long chains of GV's without intermediate variable assignments.
- Default start-when for all widgets changed to Am_Default_Widget_Start_Char which is "ANY_LEFT_DOWN" to make widgets start even if shift and control keys are down, and they work on double clicks.
- If double-click in a text object or text-input widget, the string is copied into X cut buffer, and it is put into "pending-delete" mode so the next character will delete it. Using the middle button on a text field or object while the cursor is visible will copy the current contents of the X cut buffer.

1.9.6 Bug Fixes between V1.2 and V2.0alpha

- If an interactor or widget is in a scrolling window, but is outside the visible region, clicks no longer go through to those interactors or widgets.
- Fix to make virtual window managers such as tvtwm work under Unix.
- Made interactors with Am_OTHER_WINDOWS = true work, especially for new windows created after the interactor is created. Note that the slot was renamed to be Am_MULTI_OWNERS.

- We eliminated some memory leaks from Amulet.
- New color allocation scheme under X uses a more intelligent color than black if it runs out of color cells.
- Bounding box for thick, mitred polygons works correctly
- Point_In_Part for polygons works correctly.
- Caps Lock key only affects alphabetic characters on Unix.

1.9.7 Summary of Non-Backwards Compatible Changes

1) All methods must now be declared using the Am_Define_Method_Type, Am_Define_Method_Type_Impl, and Am_Define_Method macros. Details are below.

2) All formulas when assigned into slots should not use Am_Formula::Create. No longer ever use Am_Declare_Formula: instead, just declare the type to be extern Am_Formula. Details are below.

3) Changed a number of slot names to adhere to a consistent naming scheme. Details are below.

4) Moved the functions that control interactors from the COMMAND object to the interactor itself. Thus, if you were customizing a DO_METHOD, you now set that method into the interactor itself instead of into the COMMAND of the interactor. Similarly, moved Am_CREATE_NEW_OBJECT_METHOD to interactor.

5) Changed Am_Create_New_Object_Method to take a Am_Inter_Location, because the former signature did not include the reference object.

6) The various Get routines all return an Am_Value& instead of various kinds of slot references. This should not affect very much code unless you are using advanced features.

7) New default where: Am_Inter_In_Object_Or_Part and Am_Inter_In_Text_Object_Or_Part. These try to be smart about selecting a part of a group or the object itself. If you want to select a group itself, be sure to override the default with Am_Inter_In.

8) You should no longer call Am_Initialize_Inspector in your code, because it does not need to be called on each window anymore. Instead, the inspector is initialized as part of the standard Am_Initialize.

9) The names of the UNDO handlers were changed to match the Macintosh and Windows terminology, including slot names and method names and types. In particular, UNDO_THE_UNDO changed to REDO. There are also new methods to support the selective undo and redo.

10) Changed the Am_OTHER_WINDOWS slot to be Am_MULTI_OWNERS and can be a list of objects, not just windows. Can NO LONGER be a single object: must be a list, and inter's owner (or its window) MUST also be on the list. This is to support the multi-window move-grow interactor.

11) Unnamed parts are now inherited, so you don't have to invent names for parts just to get them inherited. If you want a part to NOT be inherited, you must pass false as the second parameter to Add_Part.

12) Modified Am_Demon_Set and Am_Demon_Queue to have a more C++ like interface.

13) Interface to Am_Point_List class has entirely changed, to be more consistent with other Amulet lists.

14) Removed Am_Button_Command, Am_Scroll_Command, and Am_Text_Input_Command because they are no longer needed. Use Am_Command instead. Details are below.

1.9.7.1 Details

1) Removed Am_Call, Am_Function_Call, Am_Object_Proc type, all of the Function_Narrow routines, and the old *function* types

- Predefined types of methods include Am_Object_Method, Am_Where_Method, Am_Custom_Gridding_Method, Am_Create_New_Object_Method, Am_Text_Edit_Method, Am_Register_Command_Method, (advanced opal method types = Am_Draw_Method, Am_Invalid_Method, Am_Point_In_Method, Am_Translate_Coordinates_Method, Am_Item_Method)

- Assign the method to the slot just using the method_name (which is really a pointer to a method description structure).

- Define the methods in .h files using the type, e.g.:

```
extern Am_Where_Method Am_Inter_In;
```

- If you need a new new type, put Am_Define_Method_Type in a .h file for each new type of method (new signature of parameters or return type)

- Put Am_Define_Method_Type_Impl with same name in one .cc file (or if the Am_Define_Method_Type goes in a .cc file, put Am_Define_Method_Type_Impl just below it)

- For each method of that type, use Am_Define_Method instead of the former procedure header.

- To call the method, use code like:

```
Am_Object_Method method;
method = undo_handler.Get(Am_PERFORM_UNDO_THE_UNDO);
method.Call(undo_handler);
```

instead of Am_Call or Am_Function_Call.

- Operationally:
  - For every procedure that is used as a method, change its definition to use the Am_Define_Method macro.

- If any of your methods require a new method TYPE, use the define_method_type and define_method_type_impl macros
- Change the storing of the method in the slot to NOT have any casts and NOT have an & in front of it.
- Replace all calls of methods to use the new form.

2)

```
// in .h file:
extern Am_Formula get_window_height;
// in .cc file:
Am_Define_Formula(int, get_window_height) {
... code ...
}
...
  .Set(Am_HEIGHT, get_window_height)
```

--- Operationally: replace all Am_Formula::Create(*) with just * and replace all Am_Declare_Formula's with extern Am_Formula

3) Changed the names of all slots that had _PROC or _ACTION to be _METHOD instead, including all interactor slots. Also changed "_POSSIBLE" to _"ALLOWED" in slots names for undo handlers.

14) Do a global replace of Am_Button_Command with Am_Command. Do a global replace of Am_Scroll_Command with Am_Command. Do a global replace of Am_Text_Input_Command with Am_Command.

1.10 Formal, Legal Language

1. This License Agreement, effective as of April 1, 1996, is between: Carnegie Mellon University having a principal place of business at 5000 Forbes Avenue, Pittsburgh, PA 15213-3890 ("CMU"); and a company ("COMPANY").

2. CMU owns intellectual property rights to the computer software, electronic information and data, in all forms and versions, identified as Amulet, described in CMU Docket 96-050 ("Software"), and associated documentation ("Document"), collectively ("Program").

3. CMU grants to COMPANY, upon the terms and conditions set out below, a fully-paid, nonexclusive, world-wide, royalty-free, non-revocable, commercial license to use the Program, or any portion thereof, for any purpose, including, but not limited to, the right to grant sublicenses under CMU's patent and trade secret rights, and copyrights, including any renewals and extensions, the right to use, copy adapt, prepare derivative works of, distribute, sell, lease, or otherwise dispose of, reverse engineer, or disassemble the Software, or any portion thereof (including all subsequent editions, revisions, supplements, and versions thereof), and CMU acknowledges that COMPANY hereby grants no reciprocal rights.

4. COMPANY acknowledges that the Program is a research tool still in the development stage, that it is being supplied "as is," without any accompanying services or improvements from CMU.

5. CMU MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE, OR MERCHANTABILITY, EXCLUSIVITY OR RESULTS OBTAINED FROM SPONSOR'S USE OF ANY INTELLECTUAL PROPERTY DEVELOPED UNDER THIS AGREEMENT, NOR SHALL EITHER PARTY HERETO BE LIABLE TO THE OTHER FOR INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES SUCH AS LOSS OF PROFITS OR INABILITY TO USE SAID INTELLECTUAL PROPERTY OR ANY APPLICATIONS AND DERIVATION THEREOF. CMU DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT, OR THEFT OF TRADE SECRETS AND DOES NOT ASSUME ANY LIABILITY HEREUNDER FOR ANY INFRINGEMENT OF ANY PATENT, TRADEMARK, OR COPYRIGHT ARISING FROM THE USE OF THE PROGRAM, INFORMATION, INTELLECTUAL PROPERTY, OR OTHER PROPERTY OR RIGHTS GRANTED OR PROVIDED TO IT HEREUNDER. THE USER AGREES THAT IT WILL NOT MAKE ANY WARRANTY ON BEHALF OF CMU, EXPRESSED OR IMPLIED, TO ANY PERSON CONCERNING THE APPLICATION OF OR THE RESULTS TO BE OBTAINED WITH THE PROGRAM UNDER THIS AGREEMENT.

6. COMPANY hereby agrees to defend, indemnify and hold harmless CMU, its trustees, officers, employees, attorneys and agents from all claims or demands made against them (and any related losses, expenses or costs) arising out of or relating to COMPANY's and/or its sublicensees' use of, disposition of, or conduct regarding the Licensed Technology and/or Licensed Product including but not limited to, any claims of product liability, personal injury (including, but not limited to, death) damage to property or violation of any laws or regulations including, but not limited to, claims of active or passive negligence.

7. COMPANY agrees that it will not make any warranty on behalf of CMU, express or implied, to any person concerning the application of or the results to be obtained with the Program.

8. Title to copyright to the Program and to Document shall at all times remain with CMU, and COMPANY agrees to preserve same. COMPANY agrees not to make any copies of the Document except for COMPANY's internal use, without prior written consent of CMU. COMPANY agrees to place the appropriate copyright notice on any such copies. Nothing herein shall be deemed to grant any license or rights in any other technology owned by CMU related to the Program.

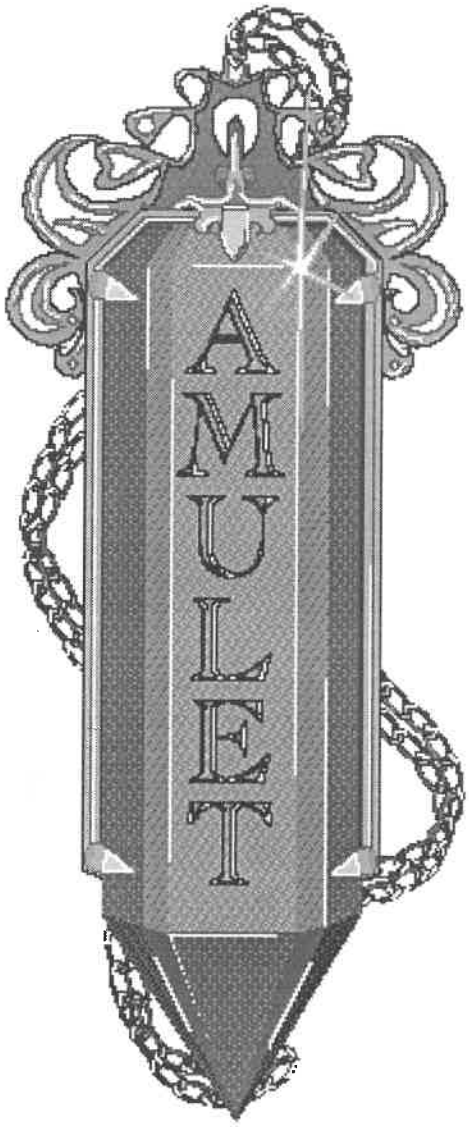
9. COMPANY owns the rights to derivative works made by or on behalf of COMPANY. Nothing herein shall be deemed to grant to CMU, or any other party, any license or any rights in any technology owned by COMPANY whether or not related to the Program, or any license or any rights to COMPANY's products whether or not incorporating any portion of the Software, or any portion of any derivative works thereof, and CMU acknowledges that CMU has no rights to same.

10. This Agreement shall be construed, interpreted and applied in accordance with the laws of the Commonwealth of Pennsylvania.

11. Nothing in this Agreement shall be construed as conferring rights to use in advertising, publicity or otherwise any trademark or the name of "CMU".

12. COMPANY understands that CMU is not responsible for support or maintenance of the Program.

The complete list of people at CMU by whom Amulet has been developed by so far is: Brad A. Myers, Alan Ferrency, Rich McDaniel, Robert C. Miller, Patrick Doane, Andy Mickish, Alex Klimovitski, Amy McGovern, William Moher, Robert Armstrong, Ashish Pimplapure, Patrick Doane, Patrick Rogan, Qiang Rao, and Chun K. So.



2. Amulet Tutorial

Amulet is a user interface development environment that makes it easier to create highly interactive, direct manipulation user interfaces in C++ for Windows NT or Unix X/11. This tutorial introduces the reader to the basic concepts of Amulet. After reading this tutorial and trying the examples with a C++ compiler, the reader will have a basic understanding of: the prototype-instance system of objects in Amulet; how to create windows and display graphical objects inside them; how to constrain the positions of objects to each other using formulas; how to use interactors to define behaviors on objects (such as selecting objects with the mouse); how to collect objects together into groups; how to use the Amulet widgets; and how to use some of the debugging tools in Amulet.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

2.1 Setting Up

2.1.1 Install Amulet in your Environment

Before beginning this tutorial, you should have already installed Amulet in your computing environment, and you should have a compiled version of the Amulet library. Instructions on how to do this can be found in the Amulet Overview, Section 1.4. This tutorial assumes that you are familiar with C++ and with the C++ development environment on your system.

In this tutorial, you will be introduced to the most commonly used parts of Amulet: the ORE Object system, Opal graphical object, Interactors, Command Objects, and Widgets. It includes code examples that can you can type in and compile yourself, along with discussions of Amulet programming techniques.

There is another programming interface to Amulet at the Gem layer (the Graphics and Events Module). By accessing the Gem layer, you can explicitly call the functions that Amulet uses to draw objects on the screen. Most Amulet users will not need to call Gem functions directly, because Amulet graphical objects redraw themselves automatically once they are added to a window. Gem is only needed by programmers who cannot get sufficient performance using the higher level interface.

2.1.2 Copy the Tutorial Starter Program

Throughout this tutorial, you will be typing and compiling code to observe its behavior. A starter program is installed with Amulet in the directory `samples/tutorial/` in Unix, or in `samples\tutorial\` in Windows NT and `samples:tutorial` in Macintosh. By following the instructions in this tutorial, you will iteratively edit and recompile this program in your local area while learning about Amulet.

Copy the `tutorial/` directory and its contents into your local filespace. You will edit this copy of the tutorial files while going through the tutorial, and not the original copy. Your copy of the directory should contain the files `Makefile` and `tutorial.cc`, if you're on a Unix platform, the files `tutorial.cpp`, `tutorial.mak`, and `tutorial.mdp` if you're on the PC, the files `tutorial.cpp and tutorial68K.proj` if you're on the Macintosh .

You should now build the initial tutorial program to make sure everything is installed correctly. In Unix, simply type make on the command line. On the PC, open `tutorial.mak` with Visual C++ 2.0, or `tutorial.mdp` with Visual C++ 4.0. You might need to add the file tutorial.cpp to the project if Visual C++ can't find it. On the Macintosh, open `tutorial68K.proj` with CodeWarrior. Next, build the project.

You should now be able to execute the `tutorial` program, which creates an empty window in the upper-left corner of the screen. Exit the program by placing the mouse in the Amulet window (and clicking in the window to make it active, if necessary) and typing the Amulet escape sequence, `META_SHIFT_F1`.

If you have trouble copying the starter program or generating the `tutorial` executable, there may be a problem with the way that Amulet was installed at your site. Consult Section 1.4 for detailed instructions about installing Amulet.

2.2 The Prototype-Instance System

Amulet provides a prototype-instance object system built on top of the C++ class-object hierarchy. C++ classes are defined at compile time, and the amount and type of data stored in a C++ object cannot change at run time. The C++ class is an abstract description of how to make an object, but contains no data by itself. In Amulet, every object is "real," and there is no underlying abstract class that describes an Amulet object at compile time. The prototype for an object in Amulet is another object, not an abstract class as in C++. All Amulet objects have the C++ type `Am_Object`.

The Amulet library provides many prototype objects which you can instantiate and customize in your programs. Examples include lines, circles, groups, windows, polygons, and so on. These prototypes have default values for all of the important properties of the object, such as size, position, and color. You can change these values in your instances of the objects, to customize their appearance and behavior. Any properties you do not customize will be inherited from the object's prototype.

While most of the objects we'll be working with in this Tutorial are graphical objects, Amulet objects are not necessarily tied down to graphical representations. Command objects and interactors are examples of nongraphical Amulet objects.

For a complete list of all of Amulet's default prototype objects, see Chapter 10, *Summary of Exported Objects and Slots*.

2.2.1 Objects and Slots

The properties of an Amulet object are stored in its *slots*, which are similar to a class's member variables in C++. A rectangle's slots contain values for its position (left, top), size (width, height), line-style, filling-style, and so on. In the following code, a rectangle is created and some of its slots are set with new values (it is not necessary to type in this code, it is just for discussion):

```
Am_Object my_rect = Am_Rectangle.Create ("my_rect")
   .Set (Am_LEFT, 20)
   .Set (Am_TOP, 20)
   .Set (Am_LINE_STYLE, Am_Black)
   .Set (Am_FILL_STYLE, Am_Red);

int my_left = my_rect.Get (Am_LEFT);     //  my_left has value 20
```

The `Set` operation sets the values of the objects' slots, and the corresponding `Get` operation retrieves them. `Set` takes a *slot key*, such as `Am_LEFT`, and a new value to store in the slot. `Get` takes a slot key and returns the value stored in the slot. A slot key is an index into the set of slots in an object.

There are many predefined slot keys used by Amulet objects, all starting with the "Am_" prefix, declared in the header file standard_slots.h. Slot keys that you create for your own use need to be explictly declared with special Amulet functions, as in:

```
Am_Slot_Key MY_SLOT = Am_Register_Slot_Name ("MY_SLOT");
```

There are many examples of setting and retrieving slot values throughout this tutorial.

An important difference between C++ classes and Amulet objects is that Amulet allows the dynamic creation of slots in objects. An Amulet program can add and remove slots from an object as needed. In C++ classes, the value of the class's data can be modified at runtime, but changing the amount of data in an object requires recompiling your code.

2.2.2 Dynamic Typing

Another difference between Amulet objects and C++ classes involves the type restrictions of the values being stored. In C++ classes, you are restricted to declaring member variables of a specific type, and you can only store data of that type in the variables. In contrast, Amulet uses *dynamic typing*, where the type of a slot is determined by the value currently stored in it. Any slot can hold any type of data, and a slot's type can change whenever a new value is set into the slot.

Amulet achieves dynamic typing by overloading the Set and Get operators. There are versions of Set and Get that handle most simple C++ types including int, float, double, char, and bool. They also handle more general types like strings, Amulet objects, functions, and void*. Other types are encapsulated in a type called Am_Wrapper, which allows C++ data structures to be stored in slots.

For example:

```
int i = obj.Get(Am_LEFT);
```

Amulet looks in the object obj and tries to find its slot Am_LEFT. If the slot exists, and its value is an integer, it is assigned to i. If the value there is *not* an integer, however, this causes an error. If you do not know what type a slot contains, you can get the slot into a generic Am_Value type or ask the slot what type it contains using obj.Get_Slot_Type (see Section 3.3).

2.2.3 Inheritance

When instances of an Amulet object are created, an inheritance link is established between the prototype and the instance. *Inheritance* allows instances to use slots in their prototypes without setting those slots in the instances themselves. For example, if we set the filling style of a rectangle to be gray, and then we create an instance of that rectangle, then the instance will also have a gray fill style.

This inheritance creates a hierarchy among the objects in the Amulet system. There is one object, `Am_Graphical_Object`, that all graphical objects are instances of. Figure 2-1 shows some of the objects in Amulet and how they fit into the inheritance hierarchy. Objects shown in bold are used by all Amulet programmers, while the others are internal, and intended to be accessed only by advanced users. The `Am_Map` and `Am_Group` objects are both special types of aggregates, and they inherit most of their properties from the `Am_Aggregate` prototype object. Some of their slots are inherited from `Am_Graphical_Object` through `Am_Aggregate`. The Widgets (the Amulet gadgets) are not pictured in this hierarchy, but most of them are instances of the `Am_Aggregate` object

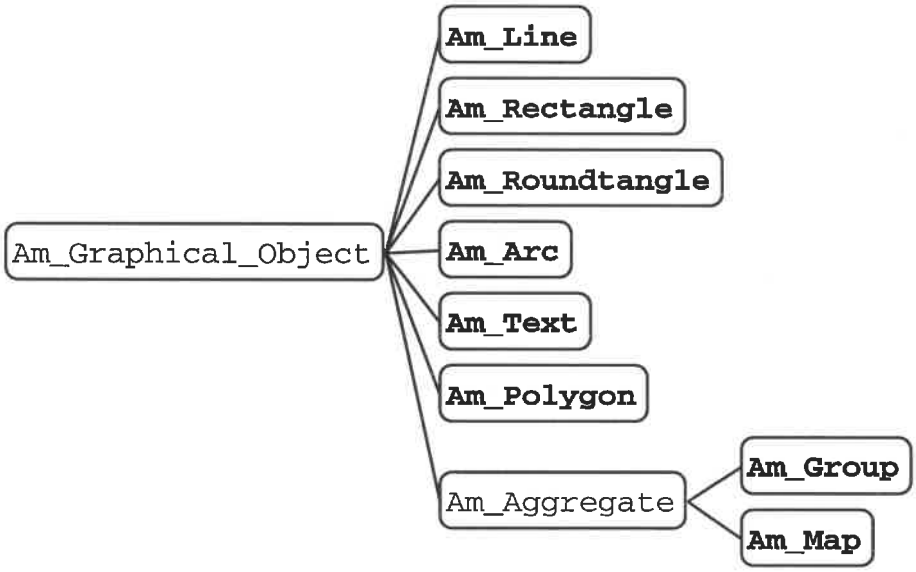


Figure 2-1: The inheritance hierarchy among some of the Amulet prototype objects.

To demonstrate inheritance, let's create an instance of a window and look at some of its inherited values. If you have followed the instructions in Section 2.1.2, you should have the file `tutorial.cc` (Unix and Mac) or `tutorial.cpp` (PC) in your local area. Edit the file. You should see:.

```
#include <amulet.h>

main (void)
{
  Am_Initialize ();

  Am_Object my_win = Am_Window.Create ("my_win")
    .Set (Am_LEFT, 20)
    .Set (Am_TOP, 50);

  Am_Screen.Add_Part (my_win);
  /* ********************************************************** */
  /* During the Tutorial, do not add or edit text below this line */
  /* ********************************************************** */

  Am_Main_Event_Loop ();
  Am_Cleanup ();
}
```

If you have not already compiled this file, do so now. In UNIX, invoke `make` in your `tutorial/` directory to generate the `tutorial` binary. On the PC and Macintosh, select "Build" from the "Project" menu. Execute `tutorial` to create a window in the upper-left corner of the screen.

The `tutorial` program creates an object called `my_win`, which is an instance of `Am_Window`. A value of 20 was installed in its `Am_LEFT` slot and 50 in its `Am_TOP` slot. These values are reflected in the position of the window on the screen.

To check that the slot values are correct, bring up the Amulet `Inspector` to examine the slots and values of the window. Move the mouse over the window and press the `F1` key. The Amulet `Inspector` will pop up a window that displays the slots and values of `my_win`, as shown in Figure 2-2 You will see many slots displayed, some of which are internal and not intended for external use. The slots with "~" in their names are internal slots. You can hide these internal slots by selecting the menu option `View: Hide Internal Slots`. Some of the other slots are "advanced" and should not be needed by most programmers. Chapter 10, *Summary of Exported Objects and Slots*, lists the primary exported slots of the main Amulet objects.

By default, object slots are sorted alphabetically by name in the Inspector. To turn this option off, choose `View: Stop Sorting By Name` from the Inspector's menu.

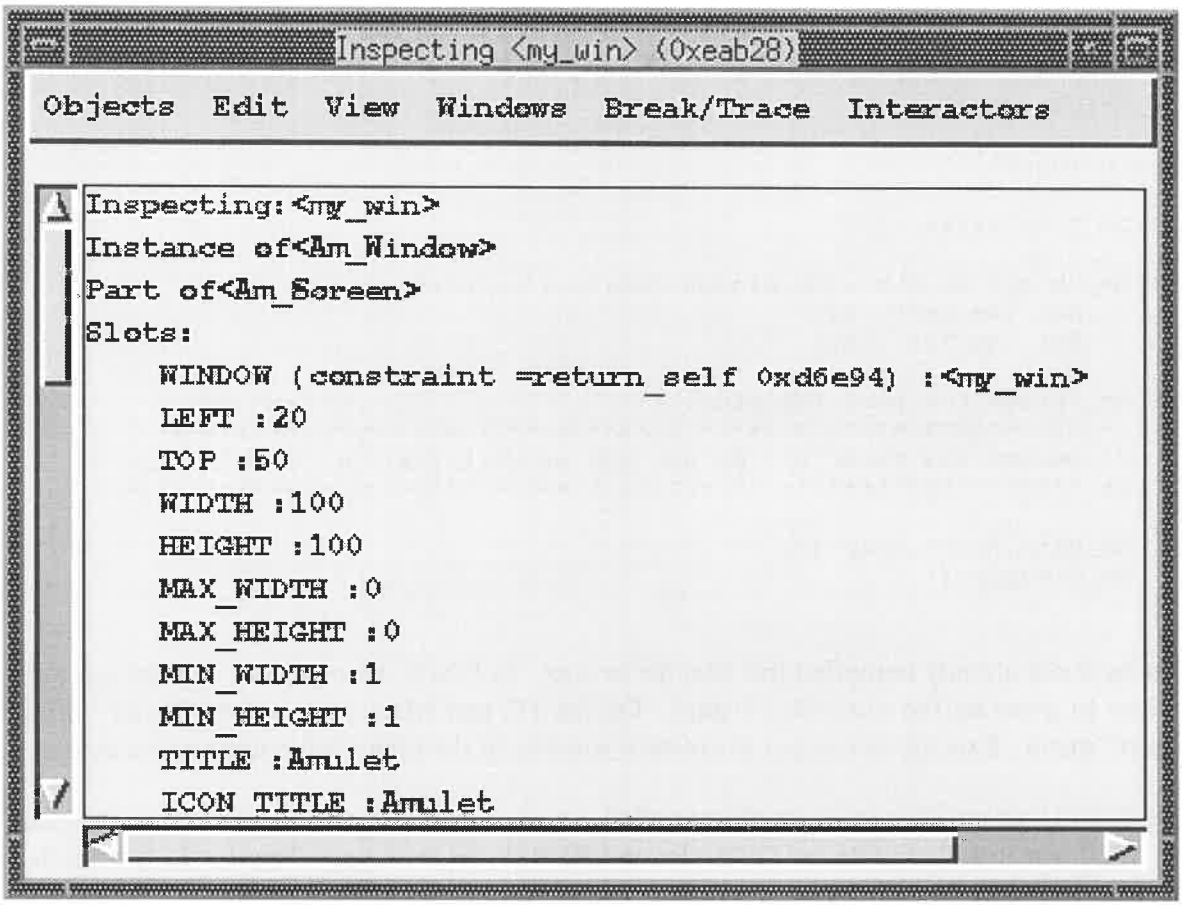


Figure 2-2: The Amulet Inspector displaying the slots and values of my_win. (The set of slots actually displayed has been abridged in this picture so that it will fit on the page.)

The `Am_LEFT` and `Am_TOP` slots of `my_win` shown in the `Inspector` contain the expected values. The `Am_WIDTH` and `Am_HEIGHT` slots contain values that were not set by the `tutorial` program. These values were *inherited* from the prototype. They were defined in the `Am_Window` object when it was created, and now `my_win` inherits those values from `Am_Window` as if you had set those slots directly into `my_win`. The `Inspector` shows they are inherited by displaying the slots in blue. Slots with local values are displayed in black. You can use the menu command `View: Hide Inherited Slots` to hide all of `my_win`'s inherited slots.

To exit the tutorial program and destroy the Amulet window, position the mouse over the window (and click to select the window, if necessary) and type `META_SHIFT_F1`. You could also choose the Inspector's `Objects: Quit Application` menu item.

Let's change the width and height of `my_win` using `Set`, the function that sets the values of slots. Edit the source code, and add the following lines immediately after the definition of `my_win`:

```
my_win.Set (Am_WIDTH, 200)
       .Set (Am_HEIGHT, 400);
```

Notice that we can cascade the calls to Set without placing semi-colons at the end of each line. Set makes this possible by returning the object that is being changed, so that the return value of Set can be used without intermediate binding. After compiling and executing the file, and hitting F1 to invoke the Inspector, you can see that we have successfully overridden the Am_WIDTH and Am_HEIGHT slots in my_win with our local values. If you move and resize the window from the window manager, the values in the inspector should change to reflect these changes as well.

The counterpart to Set is Get, which retrieves values from slots. The Inspector uses Get on my_win to obtain the values to print in the Inspector window. We can use Get directly by typing the following code into the source code, after the definition of my_win:

```
int left  = my_win.Get (Am_LEFT);
int width = my_win.Get (Am_WIDTH);
cout << "left == "  << left  << endl;
cout << "width == " << width << endl;
```

Delete the code we used to set the left, top, width, and height of the window, and see what values are printed by the cout statement when the program is run. You can see that Am_LEFT defaults to 0, and Am_WIDTH defaults to 100 if you don't set the slots explicitly.

The inheritance hierarchy shown in Figure 2-1 is traced from the leaves toward the root (from right to left) during a search for a value. Whenever we use Get to retrieve the value of a slot, the object first checks to see if it has a local value for that slot. If there is no value for the slot in the object, then the object looks to its prototype to see if it has a value for the slot. This search continues until either a value for the slot is found or the root object is reached. When no inherited or local value for the slot is found, an error is raised. This might occur if you are asking for a slot from the wrong object, or if you forget to initialize a slot's value.

2.2.4 Instances

All of the objects displayed in a window are instances of other objects. In tutorial, my_win is an instance of Am_Window. Let's create several instances of graphical objects and add them to my_win. First, make sure that your window is large enough, at least 200x200. Change your definition of my_win to look something like this:

```
Am_Object my_win = Am_Window.Create ("my_win")
  .Set (Am_LEFT, 20)
  .Set (Am_TOP, 50)
  .Set (Am_WIDTH, 200)
  .Set (Am_HEIGHT, 200);

Am_Screen.Add_Part (my_win);      // Puts my_win on the screen
```

Now we can create several graphical objects and add them to the window. Type the following code into the tutorial program after the definiton of my_win, then recompile and execute tutorial.

```
Am_Object my_arc = Am_Arc.Create ("my_arc")
  .Set (Am_LEFT, 10)
  .Set (Am_TOP, 10);

Am_Object my_text = Am_Text.Create ("my_text")
  .Set (Am_LEFT, 80)
  .Set (Am_TOP, 30)
  .Set (Am_TEXT, "This is my_text");

Am_Object my_rect = Am_Rectangle.Create ("my_rect")
  .Set (Am_LEFT, 10)
  .Set (Am_TOP, 100)
  .Set (Am_WIDTH, 180)
  .Set (Am_HEIGHT, 80)
  .Set (Am_FILL_STYLE, Am_Red);

my_win.Add_Part (my_arc)
  .Add_Part (my_text)
  .Add_Part (my_rect);
```

The circle, text, and rectangle will be displayed in the window. You can position the mouse over any of the objects and hit `F1` to display the slots of the object in the `Inspector`. If you hit `F1` while the mouse is over the background of the window, you will raise the `Inspector` for the window itself. While inspecting `my_win`, you can see at the bottom of the `Inspector` display that the new objects have been added as *parts* of the window.

Amulet supplies a large collection of objects that you can instantiate, including the basic graphical primitives like rectangles and circles, and the standard widgets like menus, buttons and scroll bars. Chapter 10, *Summary of Exported Objects and Slots*, lists the exported Amulet objects you can make instances of.

2.2.5 Prototypes

When programming in Amulet, inheritance among objects can eliminate a lot of duplicated code. If we want to create several objects that look similar, we could create each of them from scratch and copy all the values that we need into each object. However, inheritance allows us to define these objects more efficiently, by creating several similar objects as instances of a single prototype.

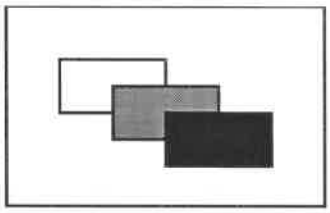


Figure 2-3: Three instances created from one prototype rectangle.

To start, look at the picture in Figure 2-3. We are going to define three rectangles with different filling styles and put them in the window. Using your current version of `tutorial`, make sure it will create a window of size at least 200x200.

Let's consider the design for the rectangles. The first thing to notice is that all of the rectangles have the same width and height. We will create a prototype rectangle which has a width of 40 and a height of 20, and then we will create three instances of that rectangle. To create the prototype rectangle, type the following.

```
Am_Object proto_rect = Am_Rectangle.Create ("proto_rect")
  .Set (Am_WIDTH, 40)
  .Set (Am_HEIGHT, 20);
```

This rectangle will not appear anywhere, because it will not be added to the window. We will create three instances of this prototype rectangle, which will be displayed. Since the prototype has the correct values for the width and height, we only need to specify the left, top, and filling styles of our instances.

```
Am_Object r1 = proto_rect.Create ("r1")
  .Set (Am_LEFT, 20)
  .Set (Am_TOP, 20)
  .Set (Am_FILL_STYLE, Am_White);

Am_Object r2 = proto_rect.Create ("r2")
  .Set (Am_LEFT, 40)
  .Set (Am_TOP, 30)
  .Set (Am_FILL_STYLE, Am_Opaque_Gray_Stipple);

Am_Object r3 = proto_rect.Create ("r3")
  .Set (Am_LEFT, 60)
  .Set (Am_TOP, 40)
  .Set (Am_FILL_STYLE, Am_Black);

my_win.Add_Part(r1)
      .Add_Part(r2)
      .Add_Part(r3);
```

When you recompile and execute, you can see that the instances `r1`, `r2`, and `r3` have inherited their width and height from `proto_rect`. You may wish to use the `Inspector` to verify this. With these three rectangles still in the window, we are ready to look at another important use of inheritance by changing values in the prototype.

Inspect `proto_rect`. You can do this by inspecting one of the three rectangles in the window, and using the right mouse button to click on `<proto_rect>` on the "`Instance of <proto_rect>`" line. Other ways to inspect this object include double left clicking on the object name, and choosing the `Objects: Inspect Object` menu item, or choosing `Objects: Inspect Object Named...` and typing "`proto_rect`" into the dialog box.

When you inspect `proto_rect`, the contents of the `Inspector` window will be replaced by the slots and values of `proto_rect`. You can bring up the new object in its own inspector window by holding down the shift key while clicking the right mouse button over `proto_rect`, or by double clicking on the object name and choosing `Objects: Inspect In New Window`.

The values of certain types of slots in the inspector can be changed by clicking on the slot's value, editing the value, and then hitting return. When you click the left mouse button in an integer or wrapper (object, style, font) value, a cursor appears and you can use standard Amulet text editing commands to change the value. Here is a brief summary of text editing commands (note: "^f" means control-f).

^f or rightarrow	forward one character
^b or leftarrow	backward one character
^a	go to beginning of line
^e	go to end of line
^h, DELETE, BACKSPACE	delete previous character
^w, ^DELETE, ^BACKSPACE	delete previous word
^d	delete next character
^u	delete entire string
^k	kill (or delete) rest of line
^y, INSERT	insert the contents of the cut buffer into the string at the current point
^c	copy the current string into the cut buffer
^g	aborts editing and returns the string to the way it was before editing started
leftdown (inside the string)	move the cursor to the specified point

All other characters go into the string (except other control characters which beep).

By editing the values in the Inspector window, change the width of proto_rect to 30 and change its height to 40. The result should look like the rectangles in Figure 2-4. Just by changing the values in the prototype rectangle, we were able to change the appearance of all its instances. This is because the three instances inherit their width and height from the prototype, even when the prototype changes.

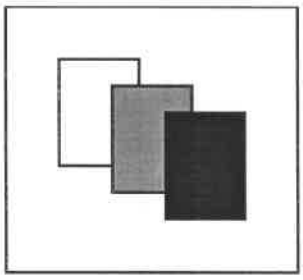


Figure 2-4: The instances change whenever the prototype object changes.

For our last look at inheritance in this section, let's override the inherited slots in one of the instances. Suppose we now want the rectangles to look like Figure 2-5. In this case, we only want to change the dimensions of one of the instances. Bring `r3` (the black rectangle) up in the `Inspector`, and change the value of its width slot to 100.

The rectangle `r3` now has its own value for its `Am_WIDTH` slot, and no longer inherits it from `proto_rect`. If you change the width of the prototype again, the width of `r3` will not be affected. However, the width of `r1` and `r2` will change with the prototype, because they still inherit the values for their `Am_WIDTH` slots. This shows how inheritance can be used flexibly to make specific exceptions to the prototype object.

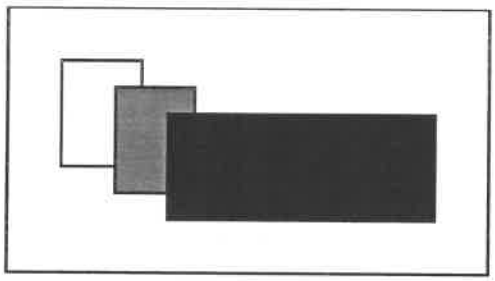


Figure 2-5: The width of `r3` is overridden by a local value, and is no longer inherited from the prototype.

2.2.6 Default Values

Because of inheritance, all instances of Amulet prototype objects have reasonable default values when they are created. As we saw in Section 2.2.4, the `Am_Window` object has its own `Am_WIDTH` value. If an instance of it is created without an explicitly defined width, the width of the instance will be inherited from the prototype. This inherited value can be considered a default value for slots in an instance. Section 10 contains a complete list of Amulet objects and the default values of their slots.

2.2.7 Destroying Objects

After objects have fulfilled their purpose, it is appropriate to destroy them. All objects occupy space in memory, and continue to do so until explicitly destroyed (or the program terminates). A `Destroy()` method is defined on all objects, so at any point in a program you can do `obj.Destroy()` to destroy *obj*.

When you destroy a graphical object (like a line or a circle), it is automatically removed from any window or group that it might be in and erased from the screen. Destroying a window or a group will destroy all of its parts. Destroying a prototype also destroys all of its instances.

2.2.8 Unnamed Objects

Sometimes you will want to create objects that do not have a particular name. Or, you might not care what the name of an object is, so you'd rather not bother thinking of a name. For example, you may want to write a function that returns a rectangle, but it will be called repeatedly and should not return multiple objects with the same name. In this case, you should allow Amulet to generate a unique name for you.

As an example, the following code creates unnamed objects and displays them in a window. Instead of supplying a quoted name to `Create`, we invoke it with no parameters.

```
Am_Object obj;
for (int i=0; i<10; i++) {
  obj = Am_Rectangle.Create()
    .Set (Am_LEFT, i*10)
    .Set (Am_TOP, i*10);
  my_win.Add_Part (obj);
}
```

When no name string is supplied to `Create`, Amulet generates a unique name for the object being created. In this case, something like `<Am_Rectangle_5>`. This name has a unique number as a suffix that prevents it from being confused with other rectangles in Amulet.

2.3 Graphical Objects

2.3.1 Lines, Rectangles, and Circles

The Opal module provides different graphical shapes including circles, rectangles, roundtangles, lines, text, bitmaps, and polygons. Each graphical object has special slots that determine its appearance, which are fully documented in chapter 4, *Opal Graphics System* and summarized in chapter 10, *Summary of Exported Objects and Slots*. Examples of creating instances of graphical objects appear throughout this tutorial.

2.3.2 Groups

In order to put a large number of objects into a window, we might create all of the objects and then add them, one at a time, to the window. However, this is usually not how we organize the objects conceptually. If we were to create a sophisticated interface with tool palettes, icons with labels, and feedback objects, we would not want to add each line and rectangle directly to the window. Instead, we would think of creating each palette from its composite rectangles, then creating the labeled icons, and then adding each assembled group to the window.

Grouping objects together like this is the function of the `Am_Group` object. Any graphical object can be part of a group - lines, circles, rectangles, widgets, and even other groups (**note:** `Am_Window` is not considered a graphical object, even though it does appear on the screen). Usually all the parts of a group are related in some way, like all the selectable icons in a tool palette.

Groups define their own coordinate system, meaning that the left and top of their parts is offset from the origin of the group. Changing the position of the group *translates* the position of all its parts. Groups also *clip* their parts to the bounding box of the group, meaning that objects outside the left, top, width, or height of the group are not drawn.

In Amulet terminology, a group is the *owner* of all of its *parts*. The `Add_Part()` and `Remove_Part()` methods are used to add and remove parts. You can optionally provide a slot key (a slot name, such as `MY_PART`) in an `Add_Part()` call. If a slot key is provided, then in addition to becoming a part of the group, the new part will be stored in that slot of the group. Parts with slot keys are always instantiated when instances of an existing group are created, and parts without a key are instantiated unless you specify otherwise. It is often convenient to provide slot keys for parts so that functions and formulas can easily access these objects in their groups.

Objects may be added directly to a window or to a group which, in turn, has been added to the window. When groups have other groups as parts, a group hierarchy is formed.

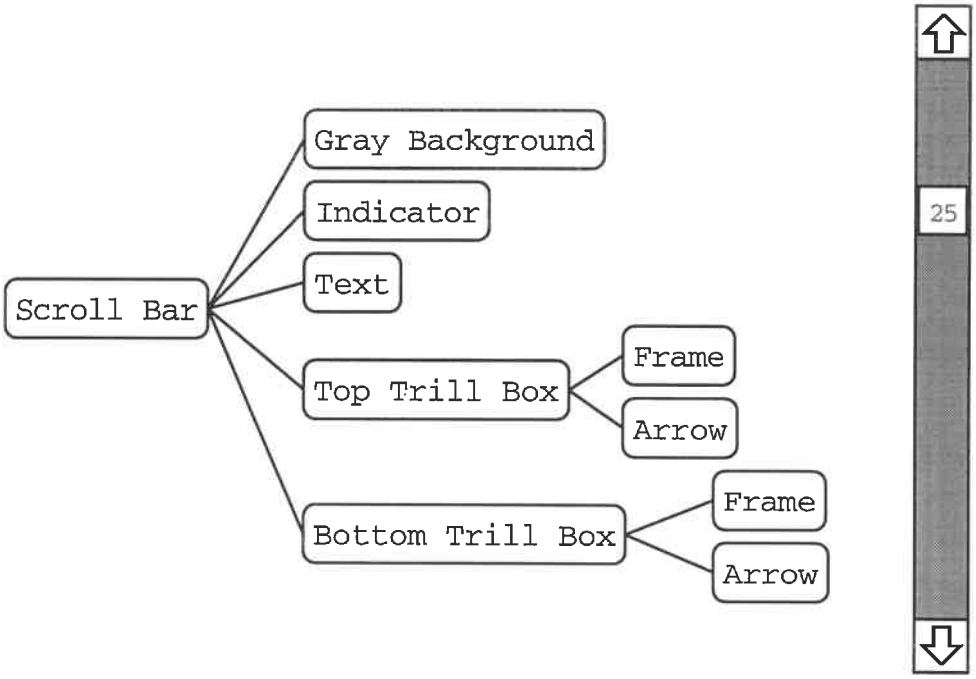


Figure 2-6: One possible hierarchy for the objects that make up a scroll bar.

In the scroll bar hierarchy, Figure 2-6, all of the leaves correspond to shapes that appear in the scroll bar. The leaves are always Amulet graphic primitives, like rectangles and text. The nodes `Top_Trill_Box` and `Bottom_Trill_Box` are both groups, each with two parts. And, of course, the top-level `Scroll_Bar` node is a group.

This group hierarchy should not be confused with the inheritance hierarchy discussed earlier. Parts of a group do not inherit values from their owners. Relationships among groups and their parts must be explicitly defined using constraints, a concept which will be discussed shortly in this tutorial.

2.3.3 Am_Group

`Am_Group` and `Am_Map` are used to form groups of other objects. They both define their own coordinate system, so that their parts are offset from the origin of the group.

You may create a group and add components to it in distinct steps, or you can use the cascading style of method invocation to perform all the Set and Add_Part operations in one expression. Here is an example of code implementing a group that contains an arc and a rectangle.

```
// Declared at the top-level, outside of main()
// You may install new slots in any object, but if they are not pre-defined Amulet slots,
// starting with the "Am_" prefix, then you must define them seperately at the top-level.
// See Section 2.2.1
Am_Slot_Key ARC_PART  = Am_Register_Slot_Name ("ARC_PART");
Am_Slot_Key RECT_PART = Am_Register_Slot_Name ("RECT_PART");
...

  // Defined inside of main()
  Am_Object my_group = Am_Group.Create ("my_group")
    .Set (Am_LEFT, 20)
    .Set (Am_TOP, 20)
    .Set (Am_WIDTH, 100)
    .Set (Am_HEIGHT, 100)
    .Add_Part(ARC_PART, Am_Arc.Create ("my_circle")
                        .Set (Am_WIDTH, 100)
                        .Set (Am_HEIGHT, 100))
    .Add_Part(RECT_PART, Am_Rectangle.Create ("my_rect")
                        .Set (Am_WIDTH, 100)
                        .Set (Am_HEIGHT, 100)
                        .Set (Am_FILL_STYLE, Am_No_Style));

  // Instances of my_group
  Am_Object my_group2 = my_group.Create ("my_group2")
    .Set (Am_LEFT, 150);

  Am_Object my_group3 = my_group.Create ("my_group3")
    .Set (Am_TOP, 150);

  // Don't forget to add the graphical objects to the window!
  my_win.Add_Part (my_group)
    .Add_Part (my_group2)
    .Add_Part (my_group3);
```

The `Add_Part()` method works like `Set`. It takes an optional slot key, and an object to install in the group. In addition to making the object an official part of the group, it is installed in the given slot in the group, if a slot key is supplied. The objects `my_circle` and `my_rect` are stored in slots `ARC_PART` and `RECT_PART` of `my_group`. The slots `ARC_PART` and `RECT_PART` are *pointer* slots because they point to other objects. These slots provide immediate access to these objects through `my_group`, which is useful when defining constraints among the objects. Once installed, the parts can be retrieved by name from the group with the methods `Get()` and `Get_Part()`.

When an instance of `my_group` is created, its parts are duplicated in the new group. Groups `my_group2` and `my_group3` have the same structure as `my_group`, but at different positions. You can explicitly specify that a part should not be duplicated in instances of its owner by providing a second boolean parameter to `Add_Part` without a slot key. `Object.Add_Part(my_part, false)` will add `my_part` as a part to `Object`, but `my_part` will not be instantiated as a part of instances of `Object`.

2.3.4 Am_Map

A map is a kind of group that has many similar parts, all generated from a single prototype. In an `Am_Map`, a single object is defined to be an *item-prototype*, and instances of this object are generated according to a set of items. See chapter 4, *Opal Graphics System*, for details and examples of maps.

2.3.5 Windows

Any object must be added to a window in order for it to be shown on the screen. Or, the object must be added to a group that, in turn, has been added to a window. All objects in a window are continually redrawn as necessary while the `Am_Main_Event_Loop()` is running (see Section 2.5.6).

As shown in previous examples, objects are added to windows using the `Add_Part()` method. Subwindows can also be attached to windows using `Add_Part()`, using exactly the same syntax for adding groups or other graphical objects.

2.4 Constraints

In the course of putting objects in a window, it is often desirable to define relationships among the objects. You might want the tops of several objects to be aligned, or you might want a set of circles to have the same center, or you may want an object to change color if it is selected. Constraints are used in Amulet to define these relationships among objects.

Constraints can be arbitrary C++ code, and can contain local variables and calls to functions. They may also have side effects on unrelated data structures with no ill effect, including setting slots and creating and destroying other Amulet objects.

Although all the examples in this section use constraints on the positions of objects, it should be clear that constraints can be defined for filling styles, strings, or any other property of an Amulet object. Many examples of constraints can be found in the following sections of this tutorial.

2.4.1 Formulas

A formula is an explicit definition of how to calculate the value for a slot. If we want to constrain the top of one object to be the same as another, then we define a formula, and put it in the `Am_TOP` slot of the dependent object. With constraints, the value of one slot always *depends* on the value of one or more other slots, and we say the formula in that slot has *dependencies* on the other slots.

An important point about constraints is that they are always automatically maintained by the system. They are evaluated once when they are first created, and then they are *re-evaluated* when any of their dependencies change. If several objects depend on the top of a certain rectangle, then all the objects will change position whenever the rectangle is moved.

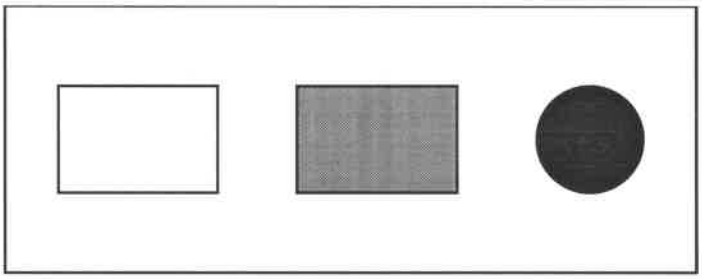


Figure 2-7: Three objects that are all aligned with the same top. The top of the gray rectangle is constrained to the white rectangle, and the top of the black circle is constrained to the top of the gray rectangle.

2.4.2 Declaring and Defining Formulas

There are several macros that are used to define formulas. These macros expand to conventional function definitions, but with special context information that Amulet uses to keep track of the constraint's dependencies. The particular macro you should use to define your formula depends on the type of the value to be returned from the formula.

`Am_Define_Formula (return_type, formula_name)` -- General purpose: returns specified `return_type`

`Am_Define_No_Self_Formula (return_type, function_name)` -- General purpose: returns specified `return_type`. Used when the formula does not reference the special `self` variable, so compiler warnings are avoided.

`Am_Define_Value_Formula (formula_name)` -- Return type is `Am_Value`

`Am_Define_Value_List_Formula (formula_name)` -- Return type is `Am_Value_List`

`Am_Define_Object_Formula (formula_name)` -- Return type is `Am_Object`

`Am_Define_Style_Formula (formula_name)` -- Return type is `Am_Style`

`Am_Define_Font_Formula (formula_name)` -- Return type is `Am_Font`

`Am_Define_Point_List_Formula (formula_name)` -- Return type is `Am_Point_List`

`Am_Define_Image_Formula (formula_name)` -- Return type is `Am_Image_Array`

To declare a formula in a header file to be exported, you should declare it of type Am_Formula. For example:

```
// inside my_file.h:
extern Am_Formula my_formula;  // my_formula is defined in my_file.cc using Am_Define_Formula()
```

2.4.3 An Example of Constraints

As our first example of defining constraints among objects, we will make the window in Figure 2-7. Let's begin by creating the white rectangle at an absolute position, and then create the other objects relative to it.

The constraints in the following examples will reference global values, and it is essential that the object variables and formulas be defined at the top-level of the program, outside of `main()`. Create the window and the first box with the following code.

```
// Defined at the top-level, outside of main()
Am_Object my_win, white_rect, gray_rect, black_arc;
...


  // Defined inside  main()

  // Create the window and display it on the screen
  my_win = Am_Window.Create ("my_win")
    .Set (Am_LEFT, 20)
    .Set (Am_TOP, 50)
    .Set (Am_WIDTH, 260)
    .Set (Am_HEIGHT, 100);
  Am_Screen.Add_Part (my_win);

  // Create the white rectangle
  white_rect = Am_Rectangle.Create ("white_rect")
    .Set (Am_LEFT, 20)
    .Set (Am_TOP, 30)
    .Set (Am_WIDTH, 60)
    .Set (Am_HEIGHT, 40)
    .Set (Am_FILL_STYLE, Am_White);

  // Add the rectangle to the window
  my_win.Add_Part (white_rect);
```

We are now ready to create the other objects that are aligned with `white_rect`. We could simply create another rectangle and a circle that each have their top at 30, but this would lead to extra work if we ever wanted to change the top of all the objects, since each object's `Am_TOP` slot would have to be changed individually. If we instead define a constraint that depends on the top of `white_rect`, then whenever the top of `white_rect` changes, the top of the other objects will automatically change, too.

Define and use a constraint that depends on the top of `white_rect` as follows:

```
// Define this at the top-level, outside of main()
Am_Define_Formula (int, top_of_white_rect) {
// The formula is named top_of_white_rect, and returns an int
  return white_rect.GV (Am_TOP);
}
...

  // Define this inside main(), after white_rect
  gray_rect = Am_Rectangle.Create ("gray_rect")
    .Set (Am_LEFT, 110)
    .Set (Am_TOP, top_of_white_rect)
    .Set (Am_WIDTH, 60)
    .Set (Am_HEIGHT, 40)
    .Set (Am_FILL_STYLE, Am_Gray_Stipple);

  my_win.Add_Part (gray_rect);
```

Without specifying an absolute position for the top of the gray rectangle, we have constrained it to always have the same top as the white rectangle. The formula in the `Am_TOP` slot of the gray rectangle was defined using the macro `Am_Define_Formula`. The `Am_Define_Formula` macro helps to define a function to be used as a constraint. The formula is named `top_of_white_rect`, and returns an `int`.

The macro `GV()` means "get value", and it is just like `Get()`, except that `GV()` causes a dependency to be established on the referenced slot, so that the formula will be reevaluated when the value in the referenced slot changes. You will usually want to use `GV()` inside of formulas, and `Get()` inside of normal functions.

To see if our constraint is working, bring up the `Inspector` on `white_rect` by hitting `F1` while the mouse is positioned over the white rectangle. Change the top of `white_rect` and notice how the gray rectangle stays aligned with its top. This shows that the formula in `gray_rect` is being reevaluated whenever its depended values change.

Now we are ready to add the black circle to the window. We have a choice of whether to constrain the top of the circle to the white rectangle or the gray rectangle. Since we are going to be examining these objects closely in the next few paragraphs, let's constrain the circle to the gray rectangle, resulting in an indirect relationship with the white one.

Define another constraint and the black circle with the following code.

```
// Define this at the top-level, outside of main()
Am_Define_Formula (int, top_of_gray_rect) {
  return gray_rect.GV (Am_TOP);
}
...

  // Define this inside main(), after gray_rect
  black_arc = Am_Arc.Create ("black_arc")
    .Set (Am_LEFT, 200)
    .Set (Am_TOP, top_of_gray_rect)
    .Set (Am_WIDTH, 40)
    .Set (Am_HEIGHT, 40)
    .Set (Am_FILL_STYLE, Am_Black);

  my_win.Add_Part (black_arc);
```

At this point, you may want to inspect the white rectangle again and change its top just to make sure the black circle follows the gray rectangle.

2.4.4 Values and constraints in slots

What happens if you set the Am_TOP of the gray rectangle now? The default for most slots, including the Am_TOP slot of Am_Rectangle, is that the new value replaces any formula in the slot. Bring up the gray rectangle in the inspector. Notice that the inspector tells you there is a constraint in the rectangle's Am_TOP slot. Change the Am_TOP of the gray rectangle by editing the value in the inspector window. You should see the grey rectangle move in the application window. Also, the inspector should no longer show a constraint in the rectangle's Am_TOP slot. The rectangle's position will not be recalculated by the constraint if white_rect moves, because the formula that was in the slot has been destroyed and replaced with a constant value.

In some slots of certain objects, such as the button widgets, there are formulas in the slots by default which are required to maintain proper behaviour of the objects. If the formulas were to be destroyed, the object would no longer work as expected. These slots have a special flag set which tells Amulet to keep the formula around even if you set the slot with a new value, and to reevaluate the formula if any of its dependencies change. Setting these slots with a new value does not replace the formula in the slot, it simply overrides the current cached value of the formula.

Any slot can be set so that formulas will not be destroyed when the slot is set. This feature is described in chapter 3, *ORE Object and Constraint System.*

2.4.5 Constraints in Groups

As mentioned in Section 2.3.3, parts can be stored in pointer slots of their group, making it easier for the parts to reference each other. Additionally, the owner is set in each part as they are added to a group. In this section, we will examine how pointer slots and variations on the GV function can be used to communicate among parts of a group.

The group we will use in this example will make the picture of concentric shapes in Figure 2-8. Suppose we want to be able to change the size and position of the shapes easily, and that this should be done by setting as few slots as possible.

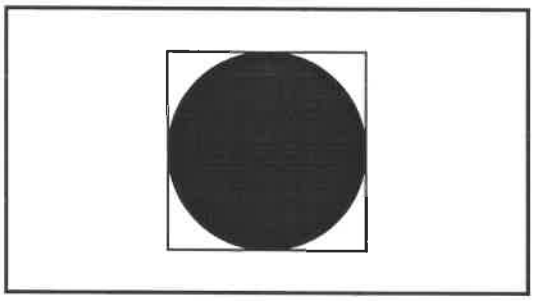


Figure 2-8: A group with two parts.

From the picture, we see that the dimensions of the rectangle are the same as the diameter of the circle. It will be helpful to put slots for the size and position at the top-level of the group, and have the parts reference these top-level values through formulas.

```
// Declared at the top-level, outside of main()
Am_Slot_Key ARC_PART  = Am_Register_Slot_Name ("ARC_PART");
Am_Slot_Key RECT_PART = Am_Register_Slot_Name ("RECT_PART");

//self is an Am_Object parameter to all formulas that holds the object the constraint is in.
// The Am_Define_Formula macro expands to define self and some other necessary variables.
Am_Define_Formula (int, owner_width) {
  return self.GV_Owner().GV(Am_WIDTH);
}

Am_Define_Formula (int, owner_height) {
  return self.GV_Owner().GV(Am_HEIGHT);
}
...

  // Defined inside of main()
  Am_Object my_group = Am_Group.Create ("my_group")
    .Set (Am_LEFT, 20)
    .Set (Am_TOP, 20)
    .Set (Am_WIDTH, 100)
    .Set (Am_HEIGHT, 100)
    .Add_Part(ARC_PART, Am_Arc.Create ("my_circle")
        .Set (Am_WIDTH, owner_width)
        .Set (Am_HEIGHT,owner_height))
    .Add_Part(RECT_PART, Am_Rectangle.Create ("my_rect")
        .Set (Am_WIDTH, owner_width)
        .Set (Am_HEIGHT, owner_height)
        .Set (Am_FILL_STYLE, Am_No_Style));
my_win.Add_Part(my_group);
```

Both parts of `my_group` get their position and dimensions from the top-level slots in `my_group`. The reference to `my_group` from the arc is through the `GV_Owner()` function, which links the part to its group. The special variable `self` is used in the formulas to reference slots within the object that the formula is installed on. The arc's left and top are relative to the origin of `my_group`, so as it inherits a position of (0,0) from the `Am_Arc` prototype, it will appear at (20,20) in the window.

Notice that the parts do not "inherit" any values from their owner. Adding parts to a group sets up a *group* hierarchy, where values travel back-and-forth over constraints, not inheritance links. If you want a part to depend on values in its owner, you have to define constraints.

The slot names for the parts could have been used to define the constraints, also. Instead of asking its owner for its dimensions, the rectangle part could have asked the arc for its dimensions. In this example the result would be the same, but here are alternate definitions for the rectangle's width and height formulas to illustrate the use of aggregate pointer slots:

```
Am_Define_Formula (int, arc_width) {
  return self.GV_Owner().GV_Part(ARC_PART).GV(Am_WIDTH);
}

Am_Define_Formula (int, arc_height) {
  return self.GV_Owner().GV_Part(ARC_PART).GV(Am_HEIGHT);
}
```

2.4.6 Common Formula Shortcuts

There are many constraints which are used very commonly, such as getting a slot value from the object's owner, or getting the value directly from another slot in the same object. There are some built in functions in Amulet to make these common constraints easier to use.

- `Am_Same_As (Am_Slot_Key key);` *// this slot gets its value from slot* `key` *in this object.*
- `Am_From_Owner (Am_Slot_Key key);` *// this slot gets its value from slot* `key` *in this object's owner.*
- `Am_From_Part (Am_Slot_Key part, Am_Slot_Key key);` *// this slot gets its value from slot* `key` *in the part of this object stored in slot* `part`.
- `Am_From_Sibling (Am_Slot_Key sibling, Am_Slot_Key key);` *// this slot gets its value from slot* `key` *in the object stored in owner's* `part` *slot.*

The following code can be used in main() to define my_group, instead of the code given above. This code does not require the two `Am_Define_Formula()` calls:

```
// Defined inside of main()
Am_Object my_group = Am_Group.Create ("my_group")
  .Set (Am_LEFT, 20)
  .Set (Am_TOP, 20)
  .Set (Am_WIDTH, 100)
  .Set (Am_HEIGHT, 100)
  .Add_Part(ARC_PART, Am_Arc.Create ("my_circle")
      .Set (Am_WIDTH,  Am_From_Owner (Am_WIDTH))
      .Set (Am_HEIGHT, Am_From_Owner (Am_HEIGHT)))
  .Add_Part(RECT_PART, Am_Rectangle.Create ("my_rect")
      .Set (Am_WIDTH,  Am_From_Sibling (ARC_PART, Am_WIDTH))
      .Set (Am_HEIGHT, Am_From_Sibling (ARC_PART, Am_HEIGHT))
      .Set (Am_FILL_STYLE, Am_No_Style));
```

2.5 Interactors

Amulet's graphical objects do not directly respond to input events. Instead, you create invisible *interactor* objects and attach them to graphical objects to respond to input. Sometimes you may just want a function to be executed when the mouse is clicked, but often you will want changes to occur in the graphics depending on the actions of the mouse. Examples include moving objects around with the mouse, editing text with the mouse and keyboard, and selecting an object from a given set.

Interactors are described in detail in chapter 5, *Interactors and Command Objects for Handling Input*, and a summary of interactors can be found in the object summary, Section 10.5. It is important to note that all of the widgets (Section 2.6 and chapter 6, *Widgets*) come with their interactors already attached. You do not need to create interactors for the widgets.

Interactors communicate with graphical objects by setting slots in the objects in response to mouse movements and keyboard keystrokes. Interactors generate side effects in the objects that they operate on. For example, the `Am_Move_Grow_Interactor` sets the left, top, width, and height slots of objects. The `Am_Choice_Interactor` sets the `Am_SELECTED` and `Am_INTERIM_SELECTED` slots to indicate when an object is currently being operated on. You might define formulas that depend on these special slots, causing the appearance of the objects (i.e., the graphics of the interface) to change in response to the mouse. The examples in the following sections show how you can use interactors this way.

Another way to use interactors (and widgets) is through their command objects (Section 2.5.5). Command objects contain methods that support undo, help, and selective enabling of operations associated with interactors and widgets. They can also contain a custom function that will be executed whenever the user operates the interactor or widget.

Figure 2-9 shows the general data flow when input events occur: the user hits a keyboard key or a mouse event, which is passed to the window manager. The Gem layer of Amulet converts it into a machine-independent form and passes it to the Interactors which finds the right interactor object to handle the event. Each interactor has an embedded command object that causes the appropriate action to take place. If this interactor is part of a widget, then the command object in the interactor calls the widget's command object. Eventually, some graphics will be modified in the Opal layer, which is automatically transformed into drawing calls at the Gem level, and then to the window manager.

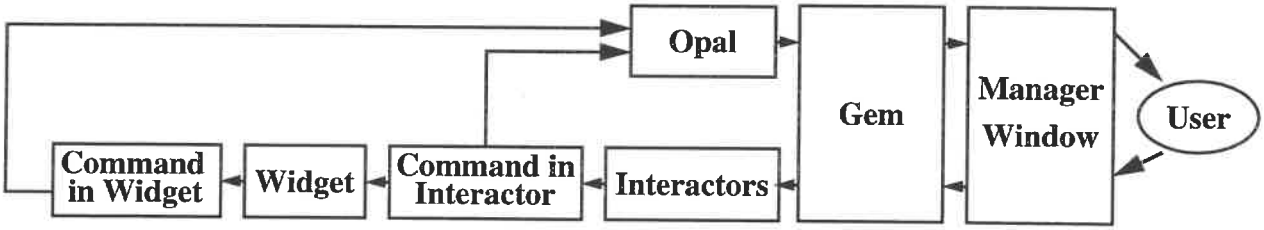


Figure 2-9: The data flow when events come from the user.

In this section we will see some examples of how to change graphics in conjunction with interactors. Section 2.7.2 describes how to use an important debugging function for interactors called `Am_Set_Inter_Trace()`. Although this tutorial only gives examples of using the `Am_One_Shot_Interactor` and `Am_Move_Grow_Interactor`, there are examples of interactors in the demo and test programs included with the Amulet files. See `samples/space/space.cc` in your Amulet source files. Instructions for compiling and running the samples are in the Overview chapter.

2.5.1 Kinds of Interactors

The design of the interactors is based on the observation that there are only a few kinds of behaviors that are typically used in graphical user interfaces. Below is a list of the available interactors.

- `Am_Choice_Interactor` - This is used to choose one or more of a set of objects. The user is allowed to move around over the objects (getting *interim feedback*) until the correct item is found, and then there will often be *final feedback* to show the final selection. The `Am_Choice_Interactor` can be used for selecting among a set of buttons or menu items, or choosing among the objects dynamically created in a graphics editor.
- `Am_One_Shot_Interactor` - This is used whenever you want something to happen immediately, for example when a mouse button is pressed over an object, or when a particular keyboard key is hit. Like the `Am_Choice_Interactor`, the `Am_One_Shot_Interactor` can be used to select among a set of objects, but it will not provide interim feedback—the one where you initially press will be the final selection. The `Am_One_Shot_Interactor` is also useful in situations where you are not selecting an object, such as when you want to get a single keyboard key.
- `Am_Move_Grow_Interactor` - This is useful in all cases where you want a graphical object to be moved or changed size with the mouse. It can be used for moving and growing objects in a graphics editor.

- `Am_New_Points_Interactor` - This interactor is used to enter new points, such as when creating new objects. For example, you might use this to allow the user to drag out a rubber-band rectangle for defining where a new rectangle should go.
- `Am_Text_Edit_Interactor` - This supports editing the text string of a text object. It supports a flexible *key translation table* mechanism so that the programmer can easily modify and add editing functions. The built-in mechanisms support basic text editing behaviors.
- `Am_Gesture_Interactor` - This interactor supports free-hand gestures, such as drawing an X over an object to delete it, or encircling a set of objects to be selected. An interactive gesture training program called Agate is provided to create new gestures for your program to use. See Section 5.3.5.6 for more information on Agate and the gesture interactor.

2.5.2 The Am_One_Shot_Interactor

In this example, we will perform an elementary operation with an interactor. We will create a window with a white rectangle inside, and then create an interactor that will make it change colors when the mouse is clicked inside of it. First, make sure you have working code that creates a window (maybe from Section 2.2.4), then add the following definitions to your program. Remember to add the rectangle to your window using `Add_Part()`.

```
// Defined at the top-level, outside of main()
Am_Define_Style_Formula (compute_fill) {
  // bool is a Boolean type defined by Amulet. Often you need to cast the value returned from GV,
  // since a slot can contain any type of object.
  if ((bool) self.GV (Am_SELECTED))
    return Am_Black;
  else
    return Am_White;
}
...

  // Defined inside main()
  Am_Object changing_rect = Am_Rectangle.Create ("changing_rect")
    .Set (Am_LEFT, 20)
    .Set (Am_TOP, 30)
    .Set (Am_WIDTH, 60)
    .Set (Am_HEIGHT, 40)
    .Set (Am_SELECTED, false)  // Set by the interactor
    .Set (Am_FILL_STYLE, compute_fill);

  my_win.Add_Part (changing_rect);
```

From the definition of the `compute_fill` formula, you can see that if the `Am_SELECTED` slot in `changing_rect` were set to `true`, then its color would turn to black. You can test this by bringing up the `Inspector` on changing_rect, and changing the value of the slot to 1. Setting the

Am_SELECTED slot is one of the side effects of the Am_One_Shot_Interactor. The following code defines an interactor which will set the Am_SELECTED slot of an object, and attaches it to changing_rect.

```
Am_Object color_inter = Am_One_Shot_Interactor.Create ("color_inter");

changing_rect.Add_Part (color_inter);
```

Now you can click on the rectangle repeatedly and it will change from white to black, and back again. From this observation, and knowing how we defined the compute_fill formula of changing_rect, you can conclude that the Am_One_Shot_Interactor is setting (and clearing) the Am_SELECTED slot of the object. This is one of the functions of this type of interactor.

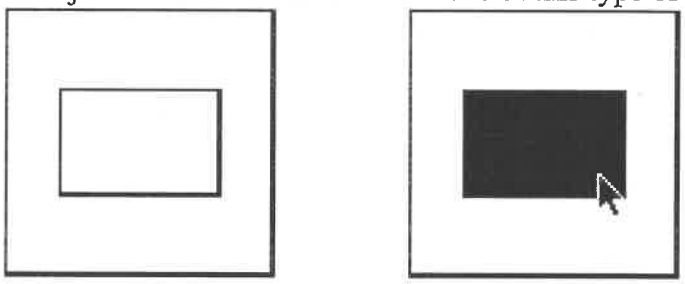


Figure 2-10: The rectangle changing_rect when its Am_SELECTED slot is false (the default), an when it is set to true by the interactor (when the mouse is clicked over it).

2.5.3 The Am_Move_Grow_Interactor

From the previous example, you can see that it is easy to change the graphics in the window using the mouse. We are now going to define several more objects in the window and create an interactor to move and grow them. The following code creates a prototype circle and several instances of it.

```
Am_Object moving_circle = Am_Arc.Create ("moving_circle")
  .Set (Am_WIDTH, 40)
  .Set (Am_HEIGHT, 40)
  .Set (Am_FILL_STYLE,  Am_No_Style);

Am_Object objs_group = Am_Group.Create ("objs_group")
  .Set (Am_WIDTH, Am_Width_Of_Parts)
  .Set (Am_HEIGHT, Am_Height_Of_Parts)
  .Add_Part (moving_circle.Create())
  .Add_Part (moving_circle.Create().Set (Am_LEFT, 50))
  .Add_Part (moving_circle.Create().Set (Am_LEFT, 100));
my_win.Add_Part(objs_group);
```

Now let's create an instance of the Am_Move_Grow_Interactor which will cause the moving circles to change position. The following interactor, when added to objs_group, works on all the parts of that group.

```
Am_Object objs_mover = Am_Move_Grow_Interactor.Create ("objs_mover");

objs_group.Add_Part(objs_mover);
```

By default, interactors try to figure out which graphical object they're supposed to manipulate. If the interactor is attached to a group-like object (`Am_Window`, `Am_Screen`, `Am_Group` or `Am_Scrolling_Group`), it looks for a part of that object to act on. Otherwise, it tests whether the mouse is directly in the object the mouse is attached to. You can change this default when you want to specify exactly what objects the interactor should operate on, by setting the interactor's `Am_START_WHERE_TEST` slot. Other methods for the the `Am_START_WHERE_TEST` are described in Section 5.3.3.2.1, or you can write your own start-where-test procedure to return the appropriate object.

Compile and run tutorial again. Now you can drag the circles around using the left mouse button. The interactor activates when you push the left button down inside any of the parts of `objs_group`. As long as you hold the button down, it moves the objects around by setting their left and top slots.

While the tutorial is running, inspect the `objs_mover` interactor. To do this, first bring up the inspector window on any of the objects on the screen. Then choose the Objects: Inspect Object Named... option from the menu, type in `objs_mover`, and hit return (or click Okay). Click in the value field of the `objs_mover`'s `Am_GROWING` slot and change the value to `1`. Now dragging the circles will cause them to change size rather than move.

2.5.4 A Feedback Object with the Am_Move_Grow_Interactor

Now let's add a feedback object to the window that will work with the moving circles. In this case, the feedback object will appear whenever we click on and try to drag a circle. The mouse will drag the feedback object, and then the real circle will move to the final position when the mouse is released.

Our feedback object will be a circle with a thick line. The `feedback_circle` object defined below will have its left, top, and visible slots set by the interactor. Given our `moving_circle` prototype, the feedback object is easy to define:

```
Am_Object feedback_circle = moving_circle.Create ("feedback_circle")
  .Set (Am_LINE_STYLE, Am_Line_8)
  .Set (Am_VISIBLE, false);

my_win.Add_Part (feedback_circle);

// The definition of the interactor, with feedback object
Am_Object objs_mover = Am_Move_Grow_Interactor.Create ("objs_mover")
  .Set (Am_START_WHERE_TEST, Am_Inter_In_Part)
  .Set (Am_GROWING, true)    // Makes the circles grow instead of move
  .Set (Am_FEEDBACK_OBJECT, feedback_circle);

objs_group.Add_Part (objs_mover);

// Don't forget to add feedback_circle and objs_mover to the right owners!
```

The Am_VISIBLE slot of feedback_circle is set to false, because we do not want it visible unless it is being used by objs_mover. The interactor will set the Am_VISIBLE slot to true and false when appropriate. Now when you move or grow the circles with the mouse, the feedback object will follow the mouse, instead of the real circle following it directly.

2.5.5 Command Objects

All interactors and widgets have command objects associated with them stored as their Am_COMMAND part. Command objects contain functions that determine what the interactor will do as it operates. For example, you can store a function in a command object that will be executed as the interactor runs in order to cause side-effects in your program. See Section 5.6 for more information on command objects inside interactors.

You can also store methods in a command object to support undo, help, and selective enabling of operations. There is a library of pre-defined command objects, so you can often use a command object from the library without writing any code. Section 6.4, *Supplied Command Objects*, describes the predefined command objects. See space and testselectionwidget for sample code that uses command objects.

Most interactors do three different things. As they run, they directly modify the associated graphical objects or feedback objects (like setting the Am_SELECTED slot). When they're finished running, they set their Am_VALUE slot and the Am_Value slot of their attached command object, and finally they call the Am_DO_METHOD of the attached command object.

To use the Am_VALUE slot of the interactor or its command object, you can establish a constraint from your object to either of these slots. If you want to make the interactor do a certain action only after it's finished running, it's best to build a custom command object. This is similar to providing a callback for the interactor to call when it's finished running. Both of these methods of using the results of an interaction are described in Section 2.6.

2.5.6 The Am_Main_Event_Loop

In order for interactors to perceive input from the mouse and keyboard, the main event loop must be running. This loop constantly checks to see if there is an event, and processes it if there is one. The automatic redrawing of graphics also relies on the main-event-loop. Exposure events, which occur when one window is uncovered or *exposed*, cause Amulet to refresh the window by redrawing the objects in the exposed area.

All Amulet programs should call two routines at the end of main(). Am_Main_Event_Loop() should be called, followed by Am_Cleanup(), which destroys the resources Amulet allocated. Your program will continue to run until Amulet perceives the escape sequence, which by default is META_SHIFT_F1. Typically, your program will have some sort of Quit button. Its do method should call Am_Exit_Main_Event_Loop(), which will cause the main-event-loop to terminate.

2.6 Widgets

The Amulet Widgets are a set of ready made gadgets that can be be added directly to a window or a group just like other graphical objects. You do not have to define separate interactors to operate the gadgets, they already have their own interactors. They have slots that can be set to customize their apperance and behavior. The Amulet Widgets are common interface building objects such as scroll bars, menus, buttons and editable text fields. Section 10.6 summarizes the widget objects, and chapter 6, *Widgets*, discusses them all in detail.

The Widgets will eventually be available in several versions, simulating the look-and-feel of the standard widgets available in the Motif, Windows, and Macintosh toolkits. Currently only the Motif style widgets have been implemented in Amulet. These widgets work on all platforms, but always look like Motif widgets. Examples that use widgets can be found in several Amulet demos, including `src/widgets/testselectionwidget` and `samples/space`. See Section 1.5 for information about the samples and demo programs.

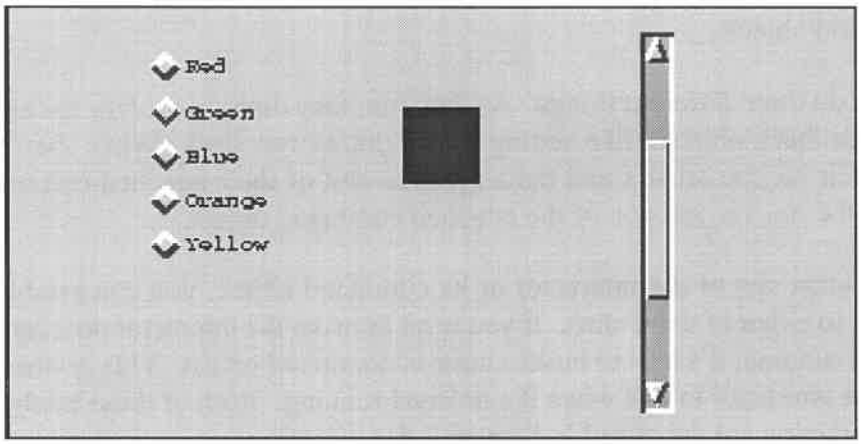


Figure 2-11: A panel of radio buttons and a vertical scroll bar, affecting a rectangle.

In this section we will use a radio button panel and a scroll bar to change the appearance of a rectangle.

There are two ways to interact with widgets. You can define a formula that depends on the value of the widget, or you can define a method to be executed by the widget's command object whenever the user activates the widget.

The code below defines the radio button panel pictured in **Figure 2-14**. Here, we define a formula for the filling style of the rectangle that depends on the value of the button panel. This formula is reevaluated every time the buttons are operated, so the rectangle changes color.

```
// Declared at the top-level, outside of main()
Am_Object color_buttons, color_rect;

// Declared at the top-level, outside of main()
Am_Define_Style_Formula (color_from_panel) {
   Am_String s = color_buttons.GV (Am_VALUE);
  if ((const char*)s) {
    if (strcmp(s, "Red") == 0) return Am_Red;
    else if (strcmp(s, "Blue") == 0) return Am_Blue;
    else if (strcmp(s, "Green") == 0) return Am_Green;
    else if (strcmp(s, "Yellow") == 0) return Am_Yellow;
    else if (strcmp(s, "Orange") == 0) return Am_Orange;
    else return Am_White;
  }
  else return Am_White;
}...

  // Defined inside main()
  color_buttons = Am_Radio_Button_Panel.Create("color_buttons")
    .Set (Am_LEFT, 10)
    .Set (Am_TOP, 10)
   .Set (Am_ITEMS, Am_Value_List ()    // An Am_Value_List supports an arbitrary list
          .Add("Red")                  //   of dynamically typed values
          .Add("Blue")
          .Add("Green")
          .Add("Yellow")
          .Add("Orange"))
    .Set (Am_FILL_STYLE, Am_Motif_Gray);

  // Defined inside main()
  color_rect = Am_Rectangle.Create("color_rect")
      .Set(Am_LEFT, 100)
      .Set(Am_TOP, 50)
      .Set(Am_WIDTH, 50)
      .Set(Am_HEIGHT, 50)
      .Set(Am_FILL_STYLE, color_from_panel);

 my_win.Add_Part (color_buttons)
   .Add_Part (color_rect);
```

Now let's create the scroll bar to change the position of the rectangle. We could define a formula that depends on the value of the scroll bar. Instead, let's use the Am_DO_METHOD of the scroll bar's command object to call a function each time the widget is operated.

```
// Defined at the top-level, outside of main()
Am_Object my_scrollbar;

// Defined at the top-level, outside of main()
Am_Define_Method(Am_Object_Method, void, my_scrollbar_do, (Am_Object cmd))
```

```
{
  int value = cmd.Get(Am_VALUE);
  color_rect.Set (Am_TOP, 20 + value);
}
...

// Defined inside main()
  my_scrollbar = Am_Vertical_Scroll_Bar.Create ("my_scrollbar")
    .Set (Am_LEFT, 250)
    .Set (Am_TOP, 10)
    .Set (Am_SMALL_INCREMENT, 5)
    .Set (Am_LARGE_INCREMENT, 20)
    .Set (Am_VALUE_1, 0)
    .Set (Am_VALUE_2, 100);

  my_scrollbar.Get_Part(Am_COMMAND).Set(Am_DO_METHOD, my_scrollbar_do);

  my_win.Add_Part (my_scrollbar);
```

`Am_Define_Method` is a macro that defines a method of an Amulet object. The first parameter is the type of the method being defined. Do methods are of type `Am_Object_Method`, meaning they take one parameter, an Amulet object, and have return type void. The second parameter to `Am_Define_Method` is the return type of the method, and then comes the method's name. Last is the method's parameter list, with an extra set of parentheses. For more information about Amulet method type declaration and method definition, see Section 3.3.8.

Compile and run the tutorial. The radio buttons will control the color of the rectangle, and the scrollbar will control its position on the screen.

2.7 Debugging

2.7.1 The Inspector

The *Inspector* is an important tool for examining properties of objects. As long as you compile with the DEBUG switch set on in your makefile or project, you will get all of the inspector code. The inspector will be automatically initialized when you call `Am_Initialize()`. While running your program, press the `F1` key over an object to inspect it.

If your keyboard does not have an `F1` key, or hitting it does not seem to do anything, you can start the `Inspector` from your program by calling the function `Am_Inspect(obj)` with the object you want to inspect as its argument. The method `obj.Text_Inspect()` prints the object's slots and values to `stdout` instead of popping up an interactive window, and is sometimes useful in a debugging environment such as `gdb`.

By default, the `Inspector` shows all of an object's inherited and local slots, sorted by name, with the inherited slots shown in blue, and the local slots shown in black. You can hide inherited slots by choosing the menu item `View: Hide Inherited Slots`. You can hide an object's internal slots (those you shouldn't modify) with the menu item `View: Hide Internal Slots`.

By defaults, the parts of an object are displayed. This can be turned off with the `View: Hide Parts` command. You can show instances of the object being displayed by choosing the `View: Show Instances` menu item.

You can edit the value of many slots in the inspector. Editing an inherited value causes the value to become local, and changes its color in the inspector from blue to black. You can edit integers, strings, Amulet Objects, Styles, Fonts, Images, and Methods from the inspector. You cannot edit Value Lists, Constraints, or parts and instances of an object.

In the `Inspector` window, clicking the right mouse button over a value that is an object will inspect that object in the same inspector window. To display an object in a new window, hold down the `SHIFT` key while pressing the right mouse button over its name in the `Inspector` window. You can specify the name of an object to inspect by using the `Objects: Inspect Object Named...` option. When you are finished with the `Inspector`, you can choose the `Objects: Done` menu item to make the current `Inspector` window disappear, or choose `Objects: Done All` if you want all of the inspector windows to be destroyed.

By default, the inspector automatically updates its contents if slots in the objects you're inspecting change. This can bog down performance when you're inspecting certain active objects. To turn off automatic refresh, choose `View: Manual Refresh`. To refresh the display in manual refresh mode, choose `Objects: Refresh Display`.

You can select slots, constraints, or objects by double clicking on their name in the Inspector. This will enable various menu items such as showing the object's prototype and instances, showing a slot's properties, or displaying a constraint's dependencies.

Sometimes you might not know exactly which object you're inspecting. Or, you might want to find out why you can't see a particular object which you think should be on the screen. With the mysterious object on the screen, choose `Objects: Flash Object`. This will cause the bounding box of the object to blink several times so you can see where it is. If you wouldn't be able to see the object flash (it's not attached to a window, it's invisible, etc.), Amulet tries to figure out why not, and prints a message to `cerr` describing why it thinks the object could not flash.

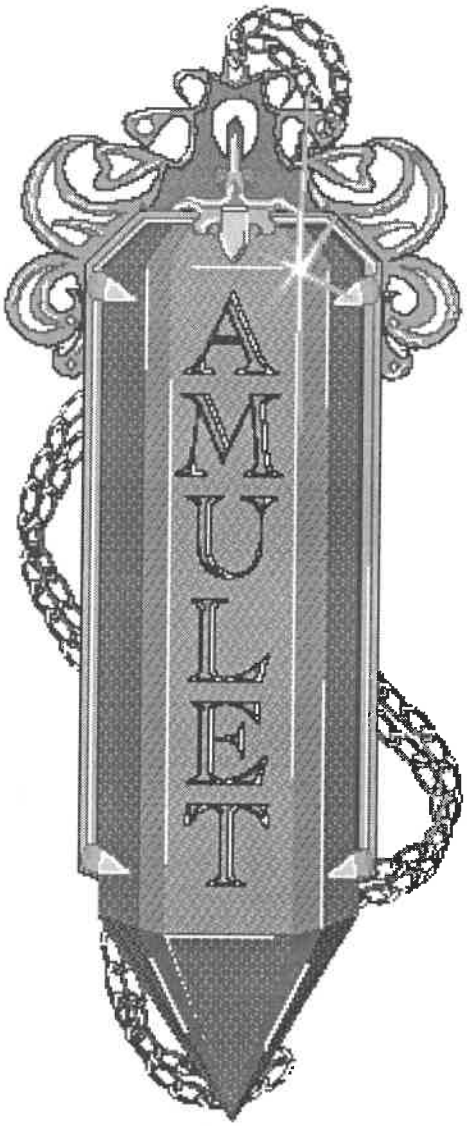
2.7.2 Tracing Interactors

The interactors and widgets provide a number of mechanisms to help programmers debug their interactions. The primary one is a *tracing* mechanism that supports printing to standard output (`cout`) whenever an "interesting" interactor event happens. Amulet supplies many options for controlling when printout occurs, as described below (full details are in the Interactors chapter). You can either set these parameters in your code and recompile, or they can be dynamically changed as your application is running, using the `Interactors` menu of the Inspector window.

The tracing choices in the Inspector's Interactors menu are:

- Turn Off Interactor Tracing — This turns off all interactor tracing.

- `Trace This Interactor` When an interactor object is selected by double clicking its name, this option will start tracing the selected interactor.
- `Trace Interactor Named...` This option brings up a dialog box prompting the user for the name of an interactor to start tracing.
- `Trace All Interactors` This starts tracing on all interactors in the application.
- `Trace Next Interactor To Run` If you don't know the name of the interactor you want to trace, this is often a useful choice. Tracing is turned on for the next interactor which starts running.
- `Trace Input Events` This prints out incoming input events, but not what happens as a result of the events. When you turn on any other tracing, Amulet automatically traces input events.
- `Trace Interactor Set Slots` This option prints a message whenever an interactor sets any slot of any object. It is useful for determining why an object's slot is being set during a particular interaction.
- `Trace Interactor Priorities` Changes to interactors' priority levels are printed.
- `Short Trace Interactors` This prints out only the names of the interactors which run, and is good for getting a general idea of what's going on in a program without all of the details.



3. ORE Object and Constraint System

This chapter describes ORE, the object and constraint level of Amulet. ORE allows programmers to create objects as instances of other objects, and define constraints among objects that keep properties consistent. For advanced users and researchers, ORE allows demons to be defined on various object operations, slot inheritance to be controlled, and even entirely new constraint solvers to be written.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

3.1 Introduction

This is the chapter for the Amulet object and constraint system, nicknamed ORE which stands for Object Registering and Encoding. This portion of the manual covers the basic operation and use of ORE and its facilities. The basic operation of ORE covers general use of objects and the kinds of values that can be stored in them. Also covered is how to make and use *formulas* that can be used to attach values together. At the end of this chapter, the means for writing new kinds of value types called wrapper types is covered.

ORE is used in Amulet as the means for representing all higher-level graphical concepts. Rectangles, for instance, are created using an exported object called `Am_Rectangle`. The process for moving a rectangle is also represented as an object. It is called `Am_Move_Grow_Interactor`. Lower-level graphical features like colors and fonts are not ORE objects in Amulet. Instead they are "wrapper types" or "wrapper values," "wrapper objects," or just plain "wrappers." What makes wrappers different from regular C++ objects is that they contain data that derives from the class `Am_Wrapper`. This makes it easy to fetch and store them in ORE objects.

The coding style for ORE objects is declarative. That means the values and behaviors of objects are specified mostly at the time an object gets created by storing initial values and declaring constraints in the needed slots. All high level objects defined in Amulet are designed to be used in a declarative way. Normal programming practice consists of choosing the kinds of objects your program needs, finding out what slots these objects contain and what the semantics of the slots do, and finally assigning values to the slots and grouping the objects together to make the final application.

Many of the concepts and processes in ORE are derived from the Garnet object system called KR. KR differs from ORE in that it was originally designed to be a artificial intelligence knowledge representation framework. The graphics came later and KR underwent an evolution that made it more compatible with the demands of a graphical system. ORE begins where KR left off. In ORE, some KR features were abandoned like multiple inheritance. Many of the good KR features that only made it into KR in the last couple releases have been put into ORE right from the start. These features include dynamic type checking and an efficient algorithm for formula propagation and evaluation. And, of course, there are many brand new features in ORE that were never part of KR. Things like the owner-part hierarchy and the ability to install multiple constraint solvers which are hoped to become very useful to Amulet programmers.

ORE features like defining new wrapper types and writing a new constraint solver are quite advanced and are covered in Section 3.11 of this chapter. These sorts of features are not necessary for the novice Amulet programmer to make working applications, but are intended to be used by system programmers or researchers that want to extend Amulet.

3.2 Include Files

The various objects, types and procedures described in this chapter are spread out though several `.h` files. Typically, one will include `amulet.h` in the code which automatically includes all the files needed. This chapter tells where various things are defined so one can look up their exact definitions. The main include files relevant to ORE are:

- `types.h`: Definitions for all the basic types including `Am_Value_Type`, `Am_Value`, `Am_Wrapper`, and `Am_Method_Wrapper`.
- `standard_slots.h`: The functions for defining slot names, and the list of Amulet-defined slots.
- `objects.h`: All of the basic object methods including slot and part iterators.
- `objects_advanced.h`: Needed for using any of the advanced features discussed in Section 3.11.
- `value_list.h`: Defines the type `Am_Value_List` and all its related methods.

For more information on Amulet include files, and how you should use them in your program, see Section 1.6 in the Overview chapter.

3.3 Objects and Slots

The Amulet object system, ORE, supports a "prototype-instance" object system. Essentially, an object is a collector for data in the form of "slots." Each slot in an object is similar to a field in a structure. Each slot stores a single piece of data for the object. A `Am_Rectangle` object, for example, has a separate slot for its left, top, width, and height.

An ORE object is different from a C++ object in many ways. The slots of ORE objects are dynamic. A program can add and remove slots as required by the given situation. Whole new types of objects can be created on demand without requiring anything to be recompiled. In C++, only the object's data can be modified and not its structure without recompiling. Furthermore in ORE, the types stored into each slot can change. For instance, the `Am_VALUE` slot can hold an integer at one time, and then a string later. ORE keeps track of the current type stored in the slot, and supports full integration with the C++ type system, including dynamic type checking.

3.3.1 Get and Set

The basic operations performed on slots are `Get` and `Set`. A slot is essentially a key–value pair. `Get` takes a slot key and returns the slot's value. `Set` takes a key and value and replaces the slot's previous value with the one given. Creating new slots is done by performing `Set` using a key that has not been used before in that object. Another way to think of an object is as a name space for slot keys. A single key can have only one value in a given object.

```
my_object.Set (Am_LEFT, 5); // Set left slot to value 5
int position = my_object.Get (Am_TOP); // Get the value of slot Am_TOP
my_object.Set (Am_ANGLE1, 45.3f); // Set the angle1 slot to float 45.3
```

Calling `Get` on a slot which does not exist raises an error. Make sure slots are initialized with values to avoid this problem. To test whether a slot exists yet, use the `Get_Slot_Type` method on objects (see Section 3.3.3), or else use the `Am_Value` form of `Get` (see Section 3.3.9).

For convenience, a special Get_Object method is available to fetch slots that are known to store a Am_Object value. This is useful for chaining together a series of Gets onto a single line without having to store into a variable or use an explicit cast. It is an error to use this method on a slot which does not store an object.

```
int i = object.Get_Object (OBJECT_SLOT).Get_Owner ().
     Get_Part (PART_SLOT).Get (Am_LEFT);
```

3.3.2 Slot Keys

A slot key in ORE is an unsigned integer. An example of a slot key is Am_LEFT. Most slot keys are defined by the Am_Standard_Slot_Keys enumeration in the file standard_slots.h. Am_LEFT turns out to be the integer 100, but one uses the name Am_LEFT because it is more descriptive. The Am_LEFT slot is used in all the graphical objects like rectangles, circles, and windows and it represents the leftmost position of that object. Potentially, the slot key 100 could be used in another object with semantics completely different from those used in graphical objects, in essence 100 could be a key besides Am_LEFT. However, ORE provides mechanisms to avoid this kind of inconsistency and makes certain that integers and slot names map one to one. The string names associated with slots are mainly used for debugging. For example, they are printed out by the inspector. The string names for slots are not used during normal program execution.

Programmers can define new slot keys for their own use by using functions defined in standard_slots.h. There are four essential functions to do this: Am_Register_Slot_Name, Am_Register_Slot_Key, Am_Get_Slot_Name, and Am_Slot_Name_Exists.

Am_Register_Slot_Name is the major function for defining new slot keys. The function returns a key which is guaranteed not to conflict with any other key chosen by this function (it is actually just a simple counter). The return value is normally stored in a global variable which is used throughout the application code. If the string name passed already has a key associated with it, Am_Register_Slot_Name will return the old key rather than allocating a new one. Thus, Am_Register_Slot_Name can also be used to look up a name to find out its current key assignment.

```
Am_Slot_Key MY_FOO_SLOT = Am_Register_Slot_Name ("My Foo Slot");
```

We recommend that programmers define their slots this way, as shown in the various example programs.

Am_Register_Slot_Key is for directly pairing a number with a name. This is useful for times when one does not want to use a global variable to store the number returned by Am_Register_Slot_Name. The number and name chosen must be known beforehand not to conflict with any other slot key chosen in the system. The range of numbers that programmers are allowed to use for their own slot keys is 10000 to 29999. Numbers outside that range are allocated for use by Amulet. The number of new slot keys needed by an application is likely to be small so running out of numbers is not likely to be a problem. The main concern will be conflicting with numbers chosen by other applications written in Amulet.

```
#define MY_BAR_SLOT 10500
Am_Register_Slot_Key (MY_BAR_SLOT, "My Bar Slot");
```

Am_NONE	The value of the slot does not exist.
Am_UNINIT	The value of the slot was set by a formula that did not have a valid value
Am_WRAPPER	A reference-counted, C++ object..
Am_OBJECT	An ORE object.
Am_INT	A signed integer.
Am_LONG	A signed long integer.
Am_BOOL	The boolean value of type `bool`.
Am_FLOAT	A single-precision floating point value.
Am_DOUBLE	A double precision floating point value.
Am_CHAR	A single character.
Am_STRING	A character string (stored as class `Am_String` [section 3.3.6], which is a form that can be readily converted into a `const char*`).
Am_VOIDPTR	`void*` in Unix and `unsigned char*` in Windows. A typedef called `Am_Ptr` can be used as a cast to make code portable between Unix and Windows.
Am_METHOD	A method. See Section 3.3.7 for a full description on defining and using methods.

The functions `Am_Get_Slot_Name` and `Am_Slot_Name_Exists` are used for testing the library of all slot keys for matches. This is especially useful when generating keys dynamically from user request.

```
const char* name = Am_Get_Slot_Name (MY_BAR_SLOT);
cout << "Slot " << MY_BAR_SLOT << " is named " << name << endl;

if (Am_Slot_Name_Exists (name)) cout << "Slot already exists\n";
```

3.3.3 Value Types

The value of `Am_LEFT` in a graphical object is an integer specifying a pixel location. Hence slot values have types, specifically the `Am_LEFT` slot has integer type in graphical objects. The type of a slot's value is determined by whatever value is stored in the slot. A slot can potentially have different types of values at different times depending on how the slot is used, but a given value has only one type so that a slot has only one type at a time. Thus, slots are "dynamically typed" like variables in Lisp.

The types supported in ORE are the majority of the simple C++ types including integer, float, double, character, and boolean. Also supported are some more high-level types like strings, ORE objects, a function type, and void pointers. Although `void*` can be used to store any type of object, ORE supports a type called `Am_Wrapper` which is used to encapsulate C++ classes and structures so that general C++ data can be stored in slots while still maintaining a degree of type checking.

```
my_object.Set (Am_LEFT, 50);
Am_Value_Type type = my_object.Get_Slot_Type (Am_LEFT);
// slot_type == Am_INT
my_object.Set (Am_FILL_STYLE, Am_Blue);
Am_Value_Type type = my_object.Get_Slot_Type (Am_FILL_STYLE);
// type == Am_Style
```

A Am_Value_Type is an unsigned short with two bit-fields. The lower 12 bits are the type base. These bits are used to distinguish individual members of a type. The upper 4 bits are the type class. Currently, there are four kinds of classes, the basic types, Am_WRAPPER, Am_METHOD, and Am_CONSTRAINT. The basic types include C++ types like Am_INT and Am_FLOAT. The Am_WRAPPER class is used to denote wrappers like Am_Object and Am_Style. Am_Method denotes types that are methods like Am_Object_Method and Am_Where_Method. Am_CONSTRAINT types are usually not stored in slots. Testing the class of a value type is performed using the macro, Am_Type_Class, which will strip off the type's base bits so that the class can be compared. A similar macro, Am_Type_Base will strip off the class bits so that the base can be compared.

```
Am_Value_Type type = object.Get_Slot_Type (Am_FILL_STYLE); // A Am_Style.
Am_Value_Type type_class = Am_Type_Class (type);
// type_class == Am_WRAPPER
```

3.3.4 The Basic Types

As shown by the examples above, the Set and Get operators are overloaded so that the normal built-in C++ primitive types can be readily used in Amulet. This section discusses some details of the primitive types, and the next few sections discuss some specialized types.

Usually, the C++ compilers can tell the appropriate types of slots from the various declarations. Thus, the compiler will correctly figure out which Set to use for each of the following:

```
my_object.Set (Am_LEFT, 50); //uses int
my_object.Set (Am_TEXT, "Foo"); //uses Am_STRING
my_object.Set (Am_PERCENT_VISIBLE, 0.75); //uses Am_FLOAT
long lng = 600000
my_object.Set (Am_VALUE_1, lng);
```

However, in some cases, the compiler cannot tell which version to use. In these cases, the programmer must put in an explicit type cast:

```
 //without cast, compiler doesn't know whether to use bool, int, void*, ...
if((bool)my_object.Get (Am_VISIBLE)) ...
 //without cast, compiler doesn't know whether to use int, long or float
int i = 5 + (int)my_object.Get(Am_LEFT);
```

Am_INT is the same as Am_LONG on Unix, Windows 95, and NT (32 bits), but on the Macintosh, an Am_INT is only 16 bits, so one must be careful to use long whenever the value might overflow 16 bits when one wants to have portable code.

The Am_Ptr type (defined in types.h) should be used wherever one would normally use a void* pointer, because Visual C++ cannot differentiate void* from some more specific pointers used by Amulet. Am_Ptr is defined as void* in Unix and unsigned char* in Windows.

3.3.5 Bools

Amulet makes extensive use of the `bool` type supplied by some C++ compilers (like gcc). For compilers that do not support it (Visual C++, ObjectCenter, etc.), Amulet defines `bool` as `int` and defines `true` as 1 and `false` as 0, so a programmer can still use `bool` in one's code. When `bool`s are supported by the compiler, Amulet knows how to return a `bool` from any kind of slot value. For example, if a slot contains a string and it is cast into a `bool`, it will return `true` if there is a string and `false` if the string is null. However, for compilers that do not support `bool`, conversion to an `int` is *not* provided, so counting on this conversion is a bad idea. Instead, it would be better to get the value into a `Am_Value` type and test that for `Valid` (see Section 3.3.9).

3.3.6 The Am_String Class

The `Am_String` type (defined in `object.h`) allows simple, null terminated C++ strings (`char*`) to be conveniently stored and retrieved from slots. It is implemented as a form of wrapper (see Section 3.3.7). An `Am_String` can be created directly from a `char*` type, likewise it can be compared directly against a `char*`. Because `Am_String` is a `Am_Wrapper` which is a reference counted structure, the programmer need not worry about the string's memory being deallocated in a local variable even if an object slot that holds a pointer to the same string gets destroyed.

The `Am_String` class will make a copy of the string if the programmer wants to modify its contents. The `Am_String` class does not allow the programmer to perform destructive modification on the string's contents.

Listed below are the basic methods defined for `Am_String`:

```
Am_String ()
Am_String (const char* initial)
```

The constructor that takes no parameters essentially creates a `NULL` char pointer. It is not equivalent to the string "". The second constructor creates the `Am_String` object with a C string as its value. The C string must be `'\0'` terminated so as to be usable with the standard string functions like `strcpy` and `strcmp`. The `Am_String` object will allocate memory to store its own copy of the string data.

```
operator const char* ()
operator char* ()
```

These casting operators make it easy to convert a `Am_String` to the more manipulable `char*` format. When a programmer casts to `const char*`, the string cannot be modified so no new memory needs to be allocated. When the programmer casts to `char*`, however, the copy of the string stored in object slots are protected by making a local copy that can be modified. The modified string can be set back to an object slot by calling `Set`.

3.3.7 Using Wrapper Types

Although one could store C++ objects into ORE slots as a `void*`, ORE provides the `Am_Wrapper` type to "wrap" C++ objects. `Am_Wrappers` provide dynamic type checking and memory management to the objects. These wrapper objects add a degree of safety to slots without sacrificing the dynamic aspects. Making new wrapper types is discussed in Section 3.11.2 and requires some practice. On the other hand, using wrapper types is simple. Notable wrapper types in Amulet are `Am_Style`, `Am_Font`, `Am_String`, `Am_Value_List` (see Section 3.8), and especially `Am_Object`, itself. Getting and setting a wrapper is syntactically identical to getting and setting an integer.

```
Am_Style blue (0.0, 0.0, 1.0); // Am_Style is a wrapper type.
my_object.Set (Am_FILL_STYLE, blue); // Using a wrapper with Set.
Am_Style color = my_object.Get (Am_FILL_STYLE); // Using a wrapper with Get.
```

A wrapper's slot type is available for testing purposes. Common wrapper types have a constant slot type such as `Am_OBJECT` and `Am_STRING`. Other wrapper types can be found using the class' static `Type_ID` method.

```
if (object.Get_Slot_Type (MY_SLOT) == Am_Style::Type_ID ())
  Am_Style color = object.Get (MY_SLOT);
```

3.3.7.1 Standard Wrapper Methods

Amulet wrappers provide a number of useful methods for querying about their state and for testing whether a given `Am_Wrapper*` belongs to a given class. These methods are common across all wrapper objects that Amulet provides. The methods are also available when programmers build their own wrapper objects using the standard macros.

The first thing that all built-in wrappers have is not a method but a special `NULL` object. The name of the `NULL` object is `Am_No_`*`Typename`* where *`typename`* is replaced by the actual name for the type. Examples are `Am_No_Font`, `Am_No_Object`, `Am_No_Value`, and `Am_No_Style`. All of the `NULL` wrapper types are essentially equivalent to a `NULL` pointer. To test whether a wrapper is `NULL` or not one uses the method `Valid()`. If a wrapper is not valid, then it should not be used to perform operations.

```
// Here the code checks to see that my_obj is not a NULL pointer by using
// the Valid method.
Am_Object my_obj = other_obj.Get (MY_OBJ);
if (my_obj.Valid ()) {
  my_obj.Set (OTHER_SLOT, 6);
}
```

Besides using the `Type_ID` method, a programmer can check the type of a value using the static `Test` method. Test takes a `Am_Value` as its parameter and returns a `bool`.

```
// Here the Test method is used to test which kind of wrapper type the
// value holds.
Am_Value_List my_list;
Am_Object my_object;
Am_Value val = obj.Get (MY_VALUE); // Am_Value is discussed later
if (Am_Value_List::Test (val))
  my_list = val;
else if (Am_Object::Test (val))
  my_object = val;
```

3.3.8 Storing Methods in Slots

ORE treats methods (procedures) stored in slots exactly the same as data. Thus, method slots can be dynamically stored, retrieved, queried and inherited like all other slots. Method types are dynamically stored just as wrapper types. Macros are provided to make defining and using methods easier.

First, to define a method whose type already exists, one uses the `Am_Define_Method` macro. In the following example, an `Am_Object_Method` is defined. An object method has a `void` type and takes a single `Am_Object` as a parameter. Other method types can have different signatures.

```
Am_Define_Method (Am_Object_Method, void, my_method,(Am_Object self))
{
  self.Set (A_SLOT, 0);
}
```

By using the macro, the compiler can check to make sure that the actual method signature matches the one defined in the type. To set the value into a slot and retrieve the typename value, one uses `Set` and `Get` in the usual way.

```
object.Set (SOME_SLOT, my_method);
Am_Object_Method hold_method = object.Get (SOME_SLOT);
```

To call a method, one invokes the `Call` field of the method's class. For instance, an `Am_Object_Method` has a `Call` field that is a procedure pointer that returns `void` and takes an `Am_Object` parameter.

```
Am_Object_Method my_method = object.Get (SOME_METHOD);
my_method.Call (some_object);
```

To check the type of method one can use its type's static `*_ID` method. The ID methods are named using the method's name so an `Am_Object_Method`'s ID is named `Am_Object_Method_ID`.

```
if (object.Get_Slot_Type (MY_SLOT) ==
    Am_Object_Method::Type_ID ()) {
  Am_Object_Method method = object.Get (MY_SLOT);
  method.Call (some_object);
}
```

Like common wrapper types, method wrappers also have a static `Test` method.

```
Am_Value value = object.Get (MY_SLOT);
if (Am_Object_Method::Test (value))
  Am_Object_Method method = value;
```

To define other method types, use the macros `Am_Define_Method_Type` and `Am_Define_Method_Type_Impl`. The first macro declares the type of the method. It is normally put into a `.h` file so that other parts of the code can use it. The `IMPL` macro is used to store the ID number of the method type. It must be stored in a `.cc` file to be compiled together with the program. In this example, a method is defined that takes two integers and returns a boolean.

```
// In the .h file
Am_Define_Method_Type (My_Method, bool, (int, int));

// In the .cc file
Am_Define_Method_Type_Impl (My_Method);
```

With these defined, a programmer can create methods of type `My_Method` and they will behave as all other ORE methods.

```
Am_Define_Method (My_Method, bool, equals, (int param1, int param2))
{
  return param1 == param2;
}

object.Set (EQUALS_SLOT, equals);
```

The procedure stored in the global method declaration can be used directly by calling its `Call` field. For example, using the `equals` method defined above, one can call:

```
bool result = equals.Call (5, 12);
```

3.3.9 Using Am_Value To Get A Slot Without Errors

Most of the time a programmer knows precisely what sort of value is stored in a slot. For these situations, the most convenient form of `Get` is the one that returns the value directly. This form has the declaration:

```
const Am_Value& Get (Am_Slot_Key key) const;
```

`Am_Value` is a union for all the ORE types. The `Am_Value` type can be coerced to all the standard Amulet types including wrappers. Normally, the programmer simply sets the return value directly into the final destination variable. But there are times when the programmer will want to call `Get` on a slot but does not know what type the slot contains (or whether the slot even exists). To determine the type, one can either query the type of the slot using the `Get_Slot_Type` method for objects, or the programmer can use the other form of `Get` that ORE provides. This form takes a `Am_Value&` as parameter instead of returning one. It has the declaration:

```
void Get (Am_Slot_Key key, Am_Value& value) const;
```

It is always valid to use the parameter style of `Get`. The return value form of `Get` will generate an error if the slot does not exist or is uninitialized. The paramter style generates an error only if it is called on an invalid or destroyed object. If the slot does not exist, the parameter form will set the `Am_Value` with type `Am_NONE`. If the slot is not initialized, then the type will be `Am_UNINIT`. (Uninitialized values concern slots with formulas [see Section 3.7].)

```
int i_value; float f_value;
Am_Value value;
my_object.Get (SOME_SLOT, value);  // Get the value regardless of type
if (value.type == Am_INT)          // The type field contains the type of value retrieved
  i_value = value;                 // Am_Value defines many casting operators
else if (value.type == Am_FLOAT)   // as assignment and constructors to aid
  f_value = value;                 // setting and retrieving the value from
                                   // the Am_Value
```

The `Am_Value` type has a number of methods, including printing (`<<`), `==`, `!=`, `Exists`, and `Valid`. `Exists` and `Valid` are used to check the contents of the `Am_Value`. `Exists` returns `true` only if the type is not `Am_NONE` or `Am_UNINIT` which would happen if the slot has no value or is uninitialized. `Valid` returns true if the value exists (as in `Exists`) and if the value is not zero as well:

```
Am_Value value;
my_object.Get (SOME_SLOT, value); // Get the value regardless of type
if (value.Exists()) {
```

```
  // then it is safe to use value, but value could still be zero
    ...
}
my_object.Get (SOME_SLOT, value); // This time we know slot must be an Am_Object
                                  // or uninitialized.
if (value.Valid ()) { // Checks both existing and value != 0
  // safe to use value.
    ...
}
```

3.4 Inheritance: Creating Objects

The inheritance style of ORE objects is prototype-instance (as opposed to C++ which is class-instance). A prototype-instance object model means that objects are used directly as the prototypes of other objects. There is no distinction between instances and classes; in essence, there are only instances. Specialization of sub-objects into new types is performed by adding slots to the sub-object or changing the contents of existing slots defined by the prototype.

Here is an example of creating an ORE object and setting some of its slots:

```
Am_Object my_rectangle = Am_Rectangle.Create ("my rect")
    .Set (Am_LEFT, 10)
    .Set (Am_TOP, 20)
    .Set (Am_WIDTH, 100)
    .Set (Am_HEIGHT, 150)
    ;
```

A major style convention in ORE is to write an object all in one expression. This is so that the programmer need not repeat the name of the object variable over and over. This works because `Set` returns the original object. The main components of the creation action involves:

- Choose a prototype object. In the above case, the prototype is `Am_Rectangle` which is defined as one of the Amulet graphical objects.
- Call the `Create` method on the prototype. The optional parameter to `Create` is a string which is used if one prints out the name of the object and can be used for other sorts of debugging. The alternative to `Create` is the `Copy` method which is described below.
- `Set` the initial values of slots. This includes making new slots if desired.

Although this manual uses the one expression convention for brevity and to familiarize programmers with its use, it would be just as correct to write out each individual `Create` and `Set` call on its own line.

```
Am_Object my_rectangle;
my_rectangle = Am_Rectangle.Create ("my rect");
my_rectangle.Set (Am_LEFT, 10);
my_rectangle.Set (Am_TOP, 20);
my_rectangle.Set (Am_WIDTH, 100);
my_rectangle.Set (Am_HEIGHT, 150);
```

Objects inherit all the slots of their prototype that are not set locally. Thus, if the `Am_Rectangle` object defines a color slot called `Am_LINE_STYLE` with a value of `Am_Black`, then `my_rectangle` will also have a `Am_LINE_STYLE` slot with the same value. If a slot is inherited, it will change value if the prototype's value changes. Thus, if `Am_LINE_STYLE` of `Am_Rectangle` is set to `Am_Blue`, then `my_rectangle`'s `Am_LINE_STYLE` will also change. However, the `Am_LEFT` of `my_rectangle` will not change if the `Am_LEFT` of `Am_Rectangle` is set because `my_rectangle` sets a local value for that slot. See Section 3.11.5 for a discussion about how a programmer can control the inheritance of slots.

The inheritance of Amulet objects' print names is dealt with slightly differently. Objects created with a name parameter to the `Create` call will keep that name. Objects created without a string parameter will get the name of their prototype plus a number appended at the end to distinguish it from the prototype.

The root of the inheritance tree is `Am_Root_Object`. Programmers will typically create instances of the pre-defined objects exported by the various Amulet files (as shown in the examples in this manual), but `Am_Root_Object` is useful if one defines application-specific objects that do not concern the Amulet toolkit per se.

The `Copy` method can also be used to make new objects. Instead of being an instance of the prototype object, a copied object will become a sibling of the object. Every slot in the prototype is copied in the same manner as in the original. If a slot is local in the original, it will be local in the copy likewise if the slot is inherited, the copy will also be inherited.

Other useful methods relevant to inheritance include:

- `Is_Instance_Of (`*`object`*`)` - Returns true if the object is an instance of the parameter. Objects are defined to be instances of themselves so `obj.Is_Instance_Of (obj)` will return `true`.
- `Get_Prototype ()` - Returns the prototype for the object.
- `Is_Slot_Inherited (`*`slot_key`*`)` - Returns `true` if the slot is not local, and the value is inherited from a prototype.
- `obj.Destroy ()` - Destroys the object and all its parts.
- `Get_Name ()` - Returns the string name of the object defined when the object was created. ORE also defines `operator<<` so name can be printed using C++ `cout` statements.
- `Print_Name (`*`ostream&`*`)` - Prints the string name of the object to the given stream. Acts like the << operator.
- `Text_Inspect ()` - This prints out all of the object's data including slots and parts to `cout`. It is commonly used for debugging and has been designed to be unlikely to crash when invoked inside a debugger.
- Objects can be tested for `==` and `!=` with other objects.
- `Remove_Slot (`*`slot_key`*`)` - Removes the slot from the object.

3.5 Destroying Objects

Objects are wrapper types which means that there is an internal reference counter within them that counts how many times the object is used. For objects which do not contain a reference to themselves, this means that simply eliminating all references to the object will destroy it. `Am_Command` objects (see the Interactor chapter, Section 5.6) are like this. A command object never stores a reference to itself so when all variables that refer to the command object are reset or destroyed, the command object will go away. Unfortunately, this kind of automatic deallocation is not guaranteed. Objects are containers that can hold arbitrary wrapper objects including references to itself. Any circularity will defeat the reference counting scheme; hence, most objects have to be explicitly destroyed.

To destroy an object, call the method `Destroy`. This method cleans out the contents of the objects making it have no slots, parts, or instances. Some operations, like `Get_Name`, will still work on a destroyed object but most methods will generate an error. `Destroy` will also destroy the object's instances and parts (parts are described in the next section). If one wants to preserve any parts of a destroyed object, one must be certain to call `Remove_From_Owner` or `Remove_Part` before the call to `Destroy` to salvage them. Note that objects that are not parts but simply stored in a slot will not be destroyed if the slot is destroyed. The vast majority of objects defined by Amulet require the programmer to call `Destroy` in order to free their memory. Forgetting to call `Destroy` is the most likely source of memory leaks.

3.6 Parts

In ORE, it is possible to make objects become *part* of another object. The subordinate object is called a "part" of the containing object which is called the "owner." The part-owner relationship is used heavily in Amulet programs.

The Opal level of Amulet defines the `Am_Window` and `Am_Group` objects which are designed to hold graphical parts (rectangles, circles, text, etc.). Thus, if you want to make a composite of graphical objects, they should be added as part of a window or group. Non-graphical objects can be made parts of any kind of object. Thus, an Interactor object can be a part of a rectangle, group, or any other object. Similarly, any kind of object that an application defines can be added as a part of any other object. For a graphical object, its "owner" will be a window or a group, but the owner of an interactor or application object can be any kind of object. Opal does not support adding graphical objects as parts of any other graphical objects (so that one cannot add a rectangle directly as a part of a circle; instead, one creates a group object and adds the rectangle and circle as parts of the group).

3.6.1 Parts Can Have Names

A very important distinction among parts is whether or not the part is named. A "named" part has a slot key. One can generate a key for a part the same way that they are generated for slots. When a part is named, it becomes possible to refer to it by that name in the method `Get_Part` (or by regular `Get`) and it also takes on other properties. If the part is unnamed, then the part cannot be accessed from the owner except through the part iterator (Section 3.9) or else by reading it from a list like the `Am_GRAPHICAL_PARTS` slot in group objects (described in the Opal chapter, Section 4.7.1).

In some ways, a named part of an object is like a slot that contains an object. A named part has a value and can have dependencies just like a slot. Parts are different from slots in that their type can only be `Am_Object` and that any particular object can only be assigned as a part to only one owner. Parts cannot be generated by a constraint and the inheritance mechanism for parts is not as sophisticated as that for slots.

New names for parts are defined the same way as new slot keys:

```
Am_Slot_Key MY_FOO_PART = Am_Register_Slot_Name ("MY_FOO_PART");
Am_Object my_obj = Am_Group.Create("My_Obj")
    .Add_Part(MY_FOO_PART, Am_Rectangle.Create("foo")); //named part
```

3.6.2 How Parts Behave With Regard To Create and Copy

When an instance is made of an object which has parts, then instances are made for each of the parts also. When an object is copied, copies are made for each part. This instancing and copying behavior can be overridden by specifying false in the `Add_Part` call when the part is added. Only unnamed parts can be set to be not inherited. Regular slots which contain objects will *share* the same object when the slot is instanced or copied.

Thus:

```
Am_Object my_obj = Am_Group.Create ("My_Obj")
  .Add_Part (MY_PART, Am_Rectangle.Create("foo")) // named part
  .Add_Part (Am_Roundtangle.Create ("bar") // unnamed part
  .Add_Part (Am_Circle.Create ("do not instance"), false)
      // unnamed part that isn't inherited
  .Set(Am_PARENT, other_object); //slot containing an object

Am_Object my_obj2 = my_obj.Create();
// my_obj2 now has a rectangle part called MY_PART which is an instance of foo.
// It also has a roundtangle part that is not named. It does not have a circle part,
// since that part was unnamed and specified not to be inherited by the false paramter
// given in the Add_Part call. The Am_PARENT slot of both my_obj and my_obj2 point
// to the same value, other_object.
```

When an object is copied using the `Copy` method, all parts are copied along with the object. Both named and unnamed parts are copied unless the part is specified not to be inherited. Uninherited parts are never copied. If a part has a string name (the string given in the `Create` call, not the slot name), the same name will be used in the copy but a number is appended to the end of the name to distinguish it from the original.

3.6.3 Other Operations on Parts

Other methods on objects relevant to parts are listed below.

- `Get_Part (part_key)` - Returns the part of an object.
- `Get_Owner ()` - Returns the owner of an object.
- `Get_Sibling (part_key)` - Equivalent to `obj.Get_Owner().Get_Part(part_name);`
- `Remove_From_Owner ()` - Removes an object from its owner.
- `Remove_Part (part_key)` - Removes the part named `part_name`.
- `Remove_Part (object)` - Removes object `part` from owner `obj`
- `Is_Part_Of (object)` - Returns true if the object is a part of the given parameter object. Objects are considered to be parts of themselves i.e. `obj.Is_Part_Of (obj)` returns true.
- `Get_Key ()` - If the object is a named part, then this will return the slot key name for the part. If it is unnamed it will return `Am_NONE` if it can be inherited and `Am_NO_INHERIT` if it cannot.

For example:

```
Am_Slot_Key RECT = Am_Register_Slot_Name ("RECT");
Am_Object my_window = Am_Window.Create ("a window")
  .Add_Part (RECT, Am_Rectangle.Create ("a rectangle"))
  .Add_Part (Am_Line.Create ("a line"));

Am_Object rect = my_window.Get_Part (RECT);
my_window.Remove_Part (RECT);
```

As mentioned above, when one destroys an object, all of its parts are destroyed also. Removing a part does not destroy the part.

3.7 Formulas

Formulas are used to connect together the values of slots. With formulas, the programmer assigns the value of one slot to be dependent on the value of other slots. When the dependent slots change, the value of the formula will be recomputed and the formula's slot will take on the computed value.

This method of computing values from dependencies is often called constraint maintenance. ORE's mechanisms for constraint maintenance are actually more general (and complicated) than the formula constraint mentioned here. The full ORE constraint mechanism will be described in a later revision of this manual. The general mechanism allows more than one constraint system to be included in the system at the same time. The formula constraint is just one of many possible constraints that may be used in ORE. For example, Amulet currently also contains a "Web" constraint used to support multi-way interactions described in Section 3.11.3.

3.7.1 Formula Functions

An ORE formula consists of a C++ function that defines the dependencies of a slot and returns the value to set into the slot. The parameter list of a formula function is always the same two parameters. The first parameter, `self`, is an `Am_Object` which points to the object containing the slot. The second parameter, `cc`, is an `Am_Constraint_Context&`. The `cc` is an opaque handle (which means that its internal representation is not visible) to the state of the formula. It is used internally by the constraint system, but is not meant to be manipulated directly by the programmer. The `cc` parameter is used to distinguish the two forms of `Get`: one which just returns the value of the slot, discussed above, the other which returns the value and also sets up a dependency link. There are also constraint versions for most of the `Get_XX` functions like `Get_Part` and `Get_Owner`. The return value of the formula function is the same type that the slot will be when it takes on the returned value.

This example creates a new formula constraint. Since it uses no macros, it looks more complicated than needed.

```
// Example of a formula function. This formula returns a value for
// Am_LEFT which will center itself within its owner's dimensions.
static int my_left_formula_proc (Am_Constraint_Context& cc, Am_Object self)
{
  Am_Object owner = self.Get_Owner (cc);
  int owner_width = owner.Get (cc, Am_WIDTH);
  int my_width = self.Get (cc, Am_WIDTH);
  return (owner_width - my_width) / 2;
}
Am_Formula my_left_formula (my_left_formula_proc, "my_left_formula");
```

The above example uses no macros so it is clear where variables are defined, and which methods take the special `cc` parameter. The global variable `my_left_formula` is the actual constraint which one would use to set a slot. Because formula functions have such a generic format, the macro `Am_Define_Formula` is usually used to save writing. Likewise, using the `cc` parameter in `Get` is common enough that macros like `GV` are available that automatically add the `cc` parameter. Using macros, the function above would look like the function below. (This particular formula definition could easily be reduced to one line.)

```
// Example of a formula function. This formula returns a value for
// Am_LEFT which will center itself within its owner's dimensions.
Am_Define_Formula (int, my_left_formula) {
  Am_Object owner = self.GV_Owner ();
  int owner_width = owner.GV (Am_WIDTH);
  int my_width = self.GV (Am_WIDTH);
  return (owner_width - my_width) / 2;
}
```

There also exists a `Set` version of `GV` called `SV`, that is used in other kinds of constraints like, in particular for constraints with multiple outputs, like `Am_Web` constraints. Though it is possible to use `SV` in a formula, it is generally easier to return the desired value. None of the formulas defined in the sample code in this section use `SV`.

There are also macros for defining formulas that return many of the built-in wrapper types. For instance, the macro `Am_Define_Style_Formula` returns the type `Am_Style`. Actually, all wrapper formulas return the same type, `Am_Wrapper*`. Since it is confusing to look at a formula function that is supposed to return `Am_Style` or `Am_Object` and see that it returns `Am_Wrapper*`, these macros are available to make the code better reflect what is really intended. The various wrapper types are described in the Opal chapter:

`Am_Define_Formula (type, formula_name)` - General purpose: returns specified `type`.

`Am_Define_No_Self_Formula (type, function_name)` - General purpose: returns specified `type`. Used when the formula does not reference the special `self` variable, so compiler warnings are avoided.

`Am_Define_Value_Formula (formula_name)` - Return type is void. This formula has a `Am_Value&` parameter named `value` which the programmer uses to return a value. Used when the formula might return different types, described in Section 3.7.1.2.

`Am_Define_Object_Formula (formula_name)` - Return type is `Am_Object`.

`Am_Define_String_Formula (formula_name)` - Return type is `Am_String`.

`Am_Define_Style_Formula (formula_name)` - Return type is `Am_Style`.

`Am_Define_Font_Formula (formula_name)` - Return type is `Am_Font`.

`Am_Define_Point_List_Formula (formula_name)` - Return type is `Am_Point_List`.

`Am_Define_Image_Formula (formula_name)` - Return type is `Am_Image_Array`.

`Am_Define_Value_List_Formula (formula_name)` - Return type is `Am_Value_List`.

`Am_Define_Cursor_Formula (formula_name)` - Return type is `Am_Cursor`.

3.7.1.1 Declaring Formulas

In order to use a formula constraint outside of the file that defines it, C++ expects an external declaration. The formula procedure does not need to externally declared. In fact formula procedures are normally declared static so that they do not infringe on the C++ external namespace. The `Am_Formula` variable is the name that needs to be externally declared. For example:

```
extern Am_Formula my_formula;
```

The type of the formula is not important in this case. All formula procedures get put into an `Am_Formula` variable that remembers the type internally.

3.7.1.2 Formulas Returning Multiple Types

Some formulas do not return only one type of value. For these formulas, the programmer cannot define a single return type. To remedy this situation, the formula constraint system can use the `Am_Value` as return value or a parameter instead of using a normal return value.

One kind of value formula declaration has return value of void and it has one extra parameter of type Am_Value& whose name is "value." The self and cc parameters are the same as in normal formulas and are used in the same way. The standard macro for declaring a multiple-type formula of this kind is Am_Define_Value_Formula. Note that this type of formula does not have return value. Instead, the value is returned by setting the Am_Value& value parameter.

```
Am_Define_Value_Formula (my_formula)
{
  if ((bool)self.GV (Am_SELECTED))
    value = 5;
  else
    value = Am_Blue;
}
```

In the above example, if the slot Am_SELECTED is true, the formula will return an integer value of 5. If it is false, it returns the color blue.

Here is an example that passes a value from a different slot and without checking what type it is:

```
Am_Define_Value_Formula (my_copy_formula) {
    other_obj.GVM (SOME_SLOT, value); // value is what is returned from
                                      // this formula
}
```

To read a slot set with a value formula the programmer can use either the Am_Value form of Get or can call Get_Slot_Type.

One can also create a formula that returns a const Am_Value as a return type:

```
Am_Define_Formula (const Am_Value, my_formula)
{
  if ((bool)self.GV (Am_SELECTED)
    return 5;
  else
    return Am_Blue;
}
```

3.7.2 Using GV

When the Get method is used with a constraint context (or equivalently, when the GV macro is used), the constraint solver decides how the actual Get is performed. The formula constraint solver sets up a dependency to the slot being fetched. Whenever the fetched slot changes value, the formula will be notified by the system and will automatically update its value by calling the formula procedure. The formula:

```
Am_Define_Formula (int, my_left) {
  int owner_width = self.GV_Owner ().GV (Am_WIDTH);
  int my_width = self.GV (Am_WIDTH);
  return (owner_width - my_width) / 2;
}
```

defines three slots as dependencies: the object's Am_OWNER slot (the GV_Owner macro expands into a GV on the Am_OWNER slot), the owner's Am_WIDTH slot, and the object's Am_WIDTH slot. A formula changes dependencies when it calls GV on different slots. For instance, the above formula's dependency on the owner's width will change if the object ever gets moved to a new owner.

A programmer can use a regular `Get` without the constraint context in a formula function, but the slot fetched will not become a dependency. Forgetting to use `GV` instead of `Get` is a common mistake for Amulet programmers. If it ever seems that a formula is not updating when it is supposed to, check to make sure that `GV` is being used in the right places.

There exists a form of `GV` for all forms of `Get` but one, `Get_Prototype`. Since the prototype of an object is fixed and can never change, there is never a need to install a dependency to it. The full list of `GV` forms is:

- `GV (slot)` - Get the specified slot, setting up a constraint.
- `GVM (s1, value)` - Get the specified slot into a `Am_Value` parameter, setting up a constraint. Unfortunately, macros cannot be overloaded like procedures; hence the different name.
- `GV_Owner ()` - Get the owner of an object, setting up a constraint.
- `GV_Part (part_name)` - Get the specified part of an object, setting up a constraint.
- `GV_Sibling (part_name)` - Get the specified sibling of this object, setting up a constraint.
- `GV_Object (slot)` - Get the value of a slot, setting up a constraint. The slot must contain an `Am_Object`.

3.7.3 Putting Formulas into Slots

To install a formula in a slot, one needs a `Am_Formula` variable. Normally, the `Am_Formula` variable will be defined using the standard macros in which case the programmer calls `Set` with the formula name to install it. If the programmer has defined the formula procedure without using the macros, then one then needs to create a formula object using that procedure. The formula object is then `Set` into a slot.

```
Am_Define_Formula (int, rect_left)
{
  return (int)self.GV_Owner ().GV (Am_WIDTH) / 2;
}

// Here the formula is used.
Am_Object my_rectangle = Am_Rectangle.Create ("my rect")
    .Set (Am_LEFT, rect_left))
    ;
```

In this example, the programmer defines a `Am_Formula` variable explicitly:

```
static int rect_left_proc (Am_Object& self, Am_Constraint_Context& cc)
{
  return (int)self.GV_Owner ().GV (Am_WIDTH) / 2;
}

// Here the formula procedure is used.
Am_Object my_rectangle = Am_Rectangle.Create ("my rect")
  .Set (Am_LEFT, Am_Formula (rect_left_proc, "name of formula"))
  ;
```

Formulas are evaluated eagerly, which means that the formula expression may be evaluated even before all the slots it depends on are defined. Since a formula cannot detect when external references are changed, it is wise to make sure that any global variable that a formula references are defined before the formula is set into a slot. Formulas that reference undefined slots or NULL objects will return the value Am_UNINIT. It is an error to fetch the value of an uninitialized slot from outside a formula so it is important that anything that a formula will depend on becomes valid as soon as possible. One can check if a slot is uninitialized by using Get_Slot_Type or fetching the slot as a Am_Value.

```
// This formula will become uninitialized if
//  a) it doesn't have an owner.
//  b) the owner's left slot doesn't exist.
//  c) the owner's left slot is uninitialized.
Am_Define_Formula (int, my_form)
{
  return self.GV_Owner ().Get (Am_LEFT);
}
```

3.7.4 Slot Setting and Inheritance of Formulas

When a slot is set, any formula that was previously in that slot will be removed by default. Just like setting a slot with a new value removes the old value of the slot, setting a slot with a value or a new formula removes the old value that was there, even if the old value was a formula.

Like values, formulas are inherited from prototypes to their instances. However, the formula in the instance might compute a value different from the formula in the prototype if the formula contains indirect links. For example, the formula that computes the width of a text object depends on the text string and font, and even though the same formula is used in every text object, most will compute different values.

Sometimes, constraints from a prototype should be retained in instances *even if the local value is set*. This requires declaring that the slot is not Single_Constraint_Mode, which is an advanced feature covered in Section 3.11.5.

3.7.5 Calling a Formula Procedure From Within Another Formula

Given a Am_Formula variable it is possible to call the formula procedure embedded within it. This process is typically done within another formula since one needs to have both a self reference and a cc parameter. Typical use for this ability is to reuse the code of another formula instead of rewriting it. An Am_Formula's procedure call method returns type const Am_Value which can be cast to the desired return type.

```
Am_Define_Formula (int, complicated_formula)
{
  // Perform some hairy computation.
}

Am_Define_Fomula (int, complicated_formula_plus_one)
{
  return (int)complicated_formula (cc, self) + 1;
}
```

3.8 Lists

Lists are widely used in Amulet. For example, many widgets require a list of labels to be displayed. The standard ORE list implementation is the `Am_Value_List` type. ORE defines this list class so that it can be a wrapper and more easily be stored as slot values. The operations on `Am_Value_Lists` are provided in the file `value_list.h`. Like slots, Lists can hold any type of value. A single list can also contain many different types of values at the same time.

Because the `Am_Value_List` is a form of wrapper, it supports all the standard wrapper operations, including:

- assignment (=), which copies the list: `Am_Value_List l2 = l1;`
- test for equality (==), which tests whether the two lists contain identical values (it iterates through each element testing for ==).
- test whether the list is valid: list.Valid ().
- test whether a value is a `Am_Value_List`: `Am_Value_List::Test (value).`

3.8.1 Current pointer in Lists

In addition to data, Am_Value_Lists contain a pointer to the "current" item. This pointer is manipulated using the following functions:

- `void Start ()` -Make first element be current. This is always legal, even if the list is empty.
- `void End ()` - Make last element be current. This is always legal, even if the list is empty.
- `void Prev ()` - Make previous element be current. This will wrap around to the end of the list when current is at the head of the list.
- `void Next ()` - Make next element be current. This will wrap around to the beginning of the list when current is at the last element in the list.
- `bool First ()` - Returns true when current element passes the first element.
- `bool Last ()` - Returns true when current element passes the last element.

The standard way to iterate through all items in a list in forward order is:

```
for (my_list.Start (); !my_list.Last (); my_list.Next ()) {
  something = my_list.Get ();
  // Use something here
}
```

Similarly, to go in reverse order, one would use:

```
for (my_list.End (); !my_list.First (); my_list.Prev ()) {
  something = my_list.Get ();
  // Use something here
}
```

Note that the pointer is *not* initialized automatically in a list, so the programmer must always call `Start` or `End` on the list before accessing its value with `Get`.

Am_Value_Lists are circular lists. Prev() and Next() wrap around, but be aware that there is a NULL list item, which you cannot do a Get() on, between the Last and First elements in the list. To wrap around, you should do an extra Next() or Prev() at the end of the list:

```
my_list.Start();
while (true) { // endless circular loop
  do_something_with(my_list.Get());
  my_list.Next();
  if (my_list.Last()) my_list.Next(); // an extra Next at the end of the list
}
```

3.8.2 Adding items to lists

There are two mechanisms for adding items to a list one element at a time: either always at the beginning or end, or at the position of the current pointer. One can also append two lists together.

To add items at the beginning or end of the list, use the Add method. Since this does not use the current pointer, one do not need to call Start. The first parameter to Add is the value to be added, which can be any primitive type, an object, a wrapper, or a Am_Value. The second parameter to Add is either Am_TAIL or Am_HEAD which defaults to Am_TAIL. This controls which end the value goes. Add returns the original Am_Value_List so multiple Adds can be chained together:

```
Am_Value_List l;
l.Add(3)
 .Add(4.0)
 .Add(Am_Rectangle.Create())
 .Add(Am_Blue)
 ;
```

To add items at the current position, the programmer must first set the current pointer. First, Start, End, Next, and Prev are used to position the current pointer as the desired location then Insert is called. The first parameter of Insert is the value to be stored, and the second parameter specifies whether the new item should go Am_BEFORE or Am_AFTER the current item. There is no default for this parameter. The current pointer does not change in this operation.

Lists can be appended with the Append method. Append is called on the lists whose elements will be at the beginning of the final result. The parameter is a list whose elements will be appended. The result is stored directly in the list on which the method is called. This method returns this just as the Add method so that it can be cascaded.

```
Am_Value_List start = Am_Value_List ().Add (1).Add (2); // list (1, 2)
Am_Value_List end = Am_Value_List ().Add (3).Add (4); // list (3, 4)
start.Append (end); // start == (1, 2, 3, 4)
```

3.8.3 Other operations on Lists

The Get method retrieves the value at the current position. Like Am_Object's Get, it returns an Am_Value which can be cast into the appropriate type. Another Get is available that returns Am_Value as a parameter. For example:

```
Am_Value v;
for (my_list.End (); !my_list.First (); my_list.Prev ()) {
  my_list.Get(v);
```

```
    cout << "List item type is " << v.type << endl << flush;
}
```

To find the type of an item without fetching the value use `Get_Type()`.

`Set` is used to change the current item in the list. This is differs from `Insert` in that it deletes the old current item and replaces it with the new value.

The `Delete` method destroys the item at the current position. It is an error to call `Delete` if there is no current element. The current pointer is shifted to the element previous to the deleted one. `Make_Empty` deletes all the items of the list.

`Am_Value_Lists` support a membership test, using the `Member` method. This starts from the current position, so **be sure to set the pointer before calling Member.** For example, to find the first instance of 3 in a list:

```
l.Start();
if (l.Member(3)) cout << "Found a 3";
```

`Member` leaves the current pointer at the position of the found item. Calling `Set` or `Delete` after a search will affect the found item. Calling `Member` again finds the next occurrence of the value.

`Length` returns the current length of the list.

`Empty` returns true if the list is empty. Note that there is a difference between being `Valid` and being `Empty`. Not being valid means that the list is `NULL` that is, it does not exist. An invalid list is also empty by definition, but an empty list is not always invalid. The list may exist and be made empty by deleting its last element, for instance. In this case, `Empty` would return `true` and `Valid` will also return `true`.

3.9 Iterators

For efficiency, ORE does not allocate an `Am_Value_List` for some types of list-like information, and instead supplies an "iterator." An iterator object contains a current pointer and allows the programmer to examine the elements one at a time. There are three kinds of iterators available in ORE: one for slots, one for instances, and another for parts. Each of the iterators has the same basic form, and the interface is essentially the same as for `Am_Value_Lists`.

3.9.1 Reading Iterator Contents

The iterator methods treat the list of items like a linked list rather than as an array. The main operations are `Start`, `Next`, and `Get`. `Start` places the iterator at the first element. `Next` moves the iterator to the next element. And `Get` returns the current element. To initialize the list, assign the iterator with the object that contains the information that the programmer want to iterate upon. For example:

```
cout << "The instances of " << my_object << " are:" << endl;
Am_Instance_Iterator iter = my_object;
for (iter.Start (); !iter.Last (); iter.Next ())  {
```

```
    Am_Object instance = iter.Get ();
    cout << instance << endl;
}
```

The first line of the example is used to initialize the iterator. The example prints out the instances of `my_object` so `my_object` is the object to assign to the iterator. The `Last` method is used to detect when the list is complete. These iterators can only be traversed in one direction.

3.9.2 Types of Iterators

The `Am_Part_Iterator` iterates over the parts of an object. Its `Get` method returns an `Am_Object` which is the part. To list the parts stored in `Am_Groups` and `Am_Windows`, however, it is better to use the `Am_Value_List` stored in the `Am_GRAPHICAL_PARTS` slot instead of the part iterator. The part iterator would list all parts in the object including many that are not graphical objects.

To iterate over the instances of an object, use an `Am_Instance_Iterator`. Its `Get` method returns an `Am_Object` which is the part. To list an object's slots (both inherited and local), use the `Am_Slot_Iterator`. Its `Get` method returns the `Am_Slot_Key` of the current slot. You can use the object method `Is_Slot_Inherited` to see if the slot is inherited or not.

3.9.3 The Order of Iterator Items

When an iterator is first initialized, there is no particular order imposed on the list. The order of the elements will be such that a single element will not repeat if the list is read from beginning to end, but the order may change if the iterator is restarted from the beginning. (Opal keeps track of the Z order [stacking or covering order] of the parts by using the `Am_GRAPHICAL_PARTS` slot which contains an `Am_Value_List` of the graphical parts, sorted correctly).

When items are added to an iterator's list while the list is being searched, the items added are not guaranteed to be seen by the iterator. The iterator may have already skipped them. The value returned by the `Length` method will be correct, but the only way to make certain all values have been seen is to restart the iterator.

Likewise, the order in which the elements are stored in each iterator is not guaranteed to be maintained when an item is *deleted* from the list. The iterators themselves cannot be used to destroy an item, but other methods like `obj.Destroy`, `obj.Remove_Part`, and `obj.Remove_Slot` will affect the contents of iterators that hold those values.

When an iterator has a slot or object as its current position, and that item gets removed, the affect on the iterator is not determined. An iterator can be restarted by calling the `Start` method in which case it will operate as expected. Though an iterator will not likely cause a crash if its current item is deleted, continued use of it could cause odd results.

For iterators that iterate over objects (specifically `Am_Part_Iterator` and `Am_Instance_Iterator`), it is possible to continue using the iterator even when items are deleted. If the programmer makes certain that the iterator does not have the deleted object as the current position when the object is removed, then the iterator will remain valid. For example:

```
// Example: Remove all parts of my_object that are instances of Am_Line.
Am_Part_Iterator iter = my_object;
iter.Start ();
while (!iter.Last ()) {
  Am_Object test_obj = iter.Get ();
  iter.Next ();
  if (test_obj.Is_Instance_Of (Am_Line))
    test_obj.Remove_From_Owner ();
}
```

In the above example, the call to `Next` occurs before the call to `Remove_From_Owner`. If these method calls were reversed, then iterator would go into an odd state and one would get undetermined results.

The `Am_Slot_Iterator` type does not have the same deletion properties as the object iterators. If a slot gets removed from an object used by a slot iterator (or the prototype of the object assuming the slot is defined there), then the affect on the iterator is undetermined. The slot iterator must be restarted whenever a slot gets added or removed from the list in order to guarantee that all slots are seen.

3.10 Errors

Whenever Amulet notices an error, it calls the `Am_Error` routine which prints out the error and then aborts the program. If you have a debugger running, it should cause the program to enter the debugger.

3.11 Advanced Features of the Object System

3.11.1 Destructive Modification of Wrapper Values

Some wrappers, like `Am_Style`'s, are immutable, which means that once created, the programmer cannot change their values. Other wrapper objects, like `Am_Value_Lists` are mutable. The default Amulet interface copies the wrapper every time it is used and automatically destroys the copies when the programmer is finished with them (explained in Section 3.11.2.2). This design prevents the programmer from accidentally changing a wrapper value that is stored in multiple places, and it helps prevent memory leaks. However, for programmers that understand how Amulet manages memory, it is unnecessarily wasteful since making copies of wrappers is not always required. This section discusses how you can modify a wrapper value without making a copy. See also the discussion of the wrapper implementation in Section 3.11.2.

When the programmer retrieves a wrapper value out of a slot, it points to the same value that is in the slot. If the value of the wrapper is changed destructively, then both the local pointer and the pointer in the slot will point to the changed value. To control this, most mutable wrappers provide a `make_unique` optional parameter in their data changing operations. The default for this parameter is `true` and makes the method create a copy of the value before it modifies it. If you call the procedure with `false`, then it will not make a copy and the value you modify will be the same as the value pointed to by all the other references. If the wrapper is pointed to in a slot, the object system will not know that the value has changed. To tell the object system that a slot has changed destructively, the programmer calls `Note_Changed` on the object after the modifications are complete. ORE responds to `Note_Changed` the same way it would respond if the slot were set with a new value. Thus, Amulet will redraw the object if necessary and notify any slots with a constraint dependent on this slot. For example:

```
obj.Make_Unique (MY_LIST_SLOT); // make sure that only one value is modified
Am_Value_List list = obj.Get (MY_LIST_SLOT);
list.Start ();
list.Delete (false); // destructively delete the first item
obj.Note_Changed (MY_LIST_SLOT); // tell Amulet that I modified the slot
```

The purpose of the `Make_Unique` call at the beginning of the previous segment concerns whether the wrapper value is unique to the slot `MY_LIST_SLOT` or not. Amulet shares wrappers between multiple slots that are not explicitly made unique. A major source of sharing occurs when an object is instantiated. For example, if one makes an instance of `obj` in the example above called `obj_instance`, then both `obj_instance` and `obj` will point to the same list in the `MY_LIST_SLOT`. In that case, the above code would modify the list for both `obj` and `obj_instance`, *but Amulet would not know about the change to* `obj_instance`. Thus, to make sure the `MY_LIST_SLOT` is unique, the programmer explicitly calls `Make_Unique` before the value of the slot is fetched. If it is beyond doubt that a wrapper value is not shared or that it is only shared in places which the programmer knows will be affected, then the `Make_Unique` call can be eliminated. `Make_Unique` will do nothing if the slot is already unique.

It is important that `Make_Unique` be called before one actually `Get`'s the value. By storing the wrapper in a local variable, the wrapper will not be unique and `Make_Unique` will make a unique copy for the slot. If the programmer would like to examine the contents of a slot before deciding whether to perform a destructive change, then one can use the `Is_Unique` method. `Is_Unique` returns a bool that tells whether the slot's value is already unique. By storing the value from `Is_Unique` before calling `Get`, the programmer can decide whether to perform destructive modification or not.

```
// Example which uses Is_Unique to guarantee the uniqueness of MY_SLOT's  value.
bool unique = obj.Is_Unique (MY_SLOT);
Am_Value_List list = obj.Get (MY_SLOT);
if (... list ...) {
  list.Add (5, Am_TAIL, !unique); // perform destructive change if unique
  if (unique)
    obj.Note_Changed (MY_SLOT);
  else
    obj.Set (MY_SLOT, list);
}
```

3.11.2 Writing a Wrapper Using Amulet's Wrapper Macros

You should consider creating a new type of wrapper whenever you need to store a C++ object into a slot of an Amulet object. A wrapper manages two things. First, it has a simple mechanism that supports dynamic type checking, so it is possible to check the type of a slot's value at run time. Second, wrappers use a reference counting scheme to prevent the value's memory from being deleted while the value is still in use.

Wrappers are created in two layers. The outermost layer is the C++ object layer used by programmers to refer to the object. The type `Am_Style` is the object layer for the `Am_Style` wrapper. Inside the object layer is the data layer. For `Am_Style`, the type is called `Am_Style_Data`. Normally, programmers are not permitted access to the data layer. The object layer of the wrapper is used to manipulate the data layer which is where the actual data for the wrapper is stored.

3.11.2.1 Creating the Wrapper Data Layer

Both the typing and reference counting is embodied by the definition of the class `Am_Wrapper` from which the data layer of all wrappers must be derived. The class `Am_Wrapper` is pure virtual with seven methods, six of which all wrappers must define. Most of the time, the programmer can use the pre-defined macros `Am_WRAPPER_DATA_DECL` and `Am_WRAPPER_DATA_IMPL` to define these six methods.

```
void Note_Reference ()
unsigned Ref_Count ()
Am_Wrapper* Make_Unique ()
void Release ()
operator== (Am_Wrapper& test_value)
Am_ID_Tag ID ()
```

Of the six methods, three are used for maintaining the reference count and making sure that only a unique wrapper value is ever modified. These are `Note_Reference`, `Make_Unique`, and `Release`. The seventh method is `void Print_Name (ostream&)` which is used to print out the value of the wrapper in a human-readable format. Unlike the other methods, this method has a default implementation though it may be defined by programmers. This method is used mostly for debugging as in the inspector or for printing out values in a property sheet. It is good to implement this method for wrappers that one intends to release for use by the public.

`Note_Reference` tells the wrapper object that the value is being referenced by another variable. The reference could be a slot or a local variable or anything else. The implementation of `Note_Reference` is normally to simply add one to the reference count. `Release` is the opposite of `Note_Reference`. It says that the variable that used to hold the value does not any longer. Typical implementation is to reduce the reference count by one. If the reference count reaches zero, then the memory should be deallocated. `Ref_Count` returns the value of the reference count so that external code can count how many of the total references it holds.

`Make_Unique` is the trickiest of these methods to understand. The basic idea is that a programmer should not be allowed to modify any wrapper value that is not unique. For example, if the programmer retrieves a `Am_Value_List` from a slot and adds an item to the list, this destructive modification should normally *not* affect the list that is still in the slot. The way to maintain this paradigm is for the method used to modify the wrapper's data to first call `Make_Unique`. If the reference count is one, the wrapper value is already unique and `Make_Unique` simply returns `this`. If the reference count is greater than one, then `Make_Unique` generates a new allocation for the value that is unique from the original and returns it. Either way only a unique wrapper value will be modified. Some wrapper types have boolean parameters on their destructive operations that turn off the behavior of `Make_Unique` to allow the programmer to do destructive modifications (see Section 3.11.1).

The `operator==` method allows the object system to compare two wrapper values against one another. The system will automatically compare the pointers so the == method must only compare the actual data. Simply returning false is sufficient for most wrappers.

The final operator is used to handle a primitive dynamic typing system. Each wrapper type is assigned a number called an `Am_ID_Tag` which is an unsigned integer. Integers are dispensed using the function `Am_Get_Unique_ID_Tag`. Normal procedure is to define a static member to the wrapper data class called `id` which gets initialized by calling `Am_Get_Unique_ID_Tag`. This function takes a name and a class ID number to generate an ID for the wrapper. ID tags and value types are one and the same concept. The wrapper ID is the same value that is stored as the wrapper's type.

All of the methods can be defined instantly by using the `Am_WRAPPER_DATA_DECL` and `Am_WRAPPER_DATA_IMPL` macros. The macros require that the user define at least two methods in the wrapper data class. The first required method is a constructor to be used by `Make_Unique`. The method is used to create a copy of the original value which can be modified without affecting the original. This can be done by making a constructor that takes a pointer to its own class as its parameter. Make sure that in all the data class constructors to initialize the `refs` member (defined by `Am_WRAPPER_DECL`) to 1. The second required method is an `operator==` to test equality. The == method does not need to check that the parameter is of the correct type because that is handled by the default implementation of the `operator==` that takes a `Am_Wrapper&` and which calls the specific == routine if the types are the same.

For example:

```
class Foo_Data : public Am_Wrapper {
    Am_WRAPPER_DATA_DECL (Foo)
  public:
    Foo_Data (Foo_Data* prev)
    {
      ... // initialize member values
      refs = 1; // Do not forget this line!
    }
    operator== (Foo_Data& test_value)
    {
      ... // compare test_value to this
    }
  protected:
    ... // define own members
};
```

```
// typically this part goes in a .cc file
  Am_WRAPPER_DATA_IMPL (Foo, (this))
```

All the standard wrapper macros take the name of the type as their first parameter. The string "_Data" is always automatically appended to the name meaning your wrapper data classes must always end in _Data. If one wants the name of the wrapper type to be Foo, the data layer type must be named Foo_Data. The Am_WRAPPER_DATA_IMPL macro takes a second parameter which is the parameter signature to use when the Make_Unique method calls the data object's constructor. In the above case, "(this)" is used because the parameter to the Foo_Data constructor is equivalent to the this pointer in the Make_Unique method. That is, the Make_Unique method will sometimes have to create a new copy of the wrapper object. The new copy will be created using one of the object's constructors. In the above case, the programmer wants to use the constructor Foo_Data(Foo_Data* prev). This constructor requires Make_Unique to pass in its this pointer as the parameter. Therefore, the parameter signature declared in the macro Am_WRAPPER_DATA_IMPL is "(this)." If the programmer wanted a different constructor to be used, the parameter set put into the macro would be different.

3.11.2.2 Using The Wrapper Data Layer

The wrapper data layer is normally manipulated only by the methods in the wrapper outer layer. One can take the Am_Foo_Data* and manipulate it as a normal C++ object with the following caveat. One must be sure that the reference count is always correct. When one uses the data pointer directly, the methods Note_Reference or Release are not being called automatically so it must be done locally in the code.

Consider the following example: The programmer wants to return a Am_Foo type but currently has a Am_Foo_Data* stored in his data.

```
// Here we move a Foo_Data pointer from one variable to another. The
// object that lives in the first variable must be released and the
// reference of the object moved to the new variable must be incremented
Foo_Data* foo_data1;
Foo_Data* foo_data2;

//... assume that foo_data1 and foo_data2 are somehow initiallized with
// real values...

if (foo_data1)
  foo_data1->Release ();
foo_data1 = foo_data2;
foo_data1->Note_Reference ();
```

To keep changes in a wrapper type local, one must call the Make_Unique method on the data before making a change to it. If the wrapper designer wants to permit the wrapper user to make destructive modifications, a boolean parameter should be added to let the user decide which to do.

```
void Foo::Change_Data (bool destructive = false)
{
  if (!destructive)
    data = data->Make_Unique ();
  data->Modify_Somehow ();
```

```
}
```

Sometimes a programmer will want to use the wrapper data pointer outside the outer wrapper layer of the object. To convert from the wrapper layer to the data layer, one uses `Narrow`.

```
Foo my_foo;
Foo_Data* data = Foo_Data::Narrow (my_foo);
data->Use_Data ();
data->Release (); // Release data when through
```

The way to test if a given wrapper is the same type as a known wrapper class is to compare IDs. The static method `TypeName_ID` is provided in the standard macros.

```
Am_Wrapper* some_wrapper_ptr = something;
Am_ID_Tag id = some_wrapper_ptr->ID ();
if (id == Foo_Data::Foo_Data_ID ())
  cout << "This wrapper is foo!" << endl;
```

Other functions provided for wrapper data classes by the standard macro are `Is_Unique`, and `Is_Zero` with are boolean functions that query the state of the reference count.

3.11.2.3 Creating The Wrapper Outer Layer

The standard macros for building the wrapper outer layer assume that the class for the wrapper data is called `TypeName_Data` where *TypeName* is the name for the wrapper outer layer. Like the data layer macros, there are two outer layer macros, one for the class declaration part and one for the implementation.

```
// Building the outer layer of Foo. This definition normally goes in
// a .h file.
class Foo { // Note there is no subclassing.
  Am_WRAPPER_DECL (Foo)
 public:
  Foo (params);
  Use ();
  Modify ();
};

  // This normally goes in the .cc file.
  Am_WRAPPER_IMPL (Foo)
```

The wrapper outer layer is given a single member named `data` which is a pointer to the data layer. In all the wrapper methods, one performs operations on the `data` member.

```
// A Foo constructor - initializes the data member.
Foo::Foo (params)
{
  data = new Foo_Data (params);
}
```

For methods that modify the contents of the wrapper data, one must be sure that the data is unique. One uses the `Make_Unique` method to manage uniqueness.

```
Foo::Modify ()
{
```

```
  if (!data)
    Am_Error ("not initialized!");
  data = data->Make_Unique ();
  data->Modify ();
}
```

Methods that do not modify data do not need to force uniqueness so they can use the wrapper data directly.

```
Foo::Use ()
{
  if (!data)
    Am_Error ("not initialized!");
  data->Use ();
}
```

Somewhere in the code, the wrapper will actually do something: calculate an expression, draw a box, whatever. Whether the programmer puts the implementation in the data layer or the outer layer is not important. Most wrapper implementations will put define their function wherever it is most convenient.

Putting a wrapper around an existing C++ class is not difficult. One can make the original class a piece of data for the wrapper data layer. If the programmer does not want to reimplement all the methods that come with the existing class, one provides a single method that returns the original class and calls the methods on that. Be certain that `Make_Unique` is called before the existing object is returned. If the object can be destructively modified, then the wrapper must be made safe before the modifications occur. However, if the programmer wants the wrapper object to behave as if it were the original class then some reimplementation may be required.

3.11.3 The Am_Web Constraint

The `Am_Web` constraint is a multi-way constraint solver. That means it can store values to more than one slot at a time. On the other hand, the one-way `Am_Formula` constraint can only store values to one slot, the slot in which the formula is stored. Multi-way constraints tend to be more difficult to use and understand than one-way constraints, which is `Am_Web` is described here in the advanced section. One would use a web instead of a formula to build a constraint that involves tying several slots together so that their values act as a unit. Also, sometimes it is more efficient to make a single web where many formulas would be required to accomplish the same task. For example, Opal uses a web to tie together the left, top, width, height slots with the x1, y1, x2, y2 slots in the `Am_Line` object. Finally, the web provides information about which dependencies have been changed during re-validation which may be required to implement certain kinds of constraints.

A web constraint consists of three procedures as opposed to one procedure in a formula. The types of these procedures are `Am_Web_Create_Proc`, `Am_Web_Initialize_Proc`, and `Am_Web_Validate_Proc`. These procedures always use the same signature since there is no return type to worry about. The validation procedure is the most similar to the formula's procedure. It is executed everytime the web is reevaluated and it typically consists of the most code. The initialization procedure is run once when the web is first created. Its purpose is to set up the web's depen-

dencies, especially the output dependencies. The create procedure concerns what happens when a slot containing a web constraint is copied or instantiated. Since a web can have multiple output dependencies, it does not really belong to any one slot. So, a slot is chosen by the programmer to be the "primary slot." That slot is considered to be the web's owner and the location where a new web will be instantiated. The create procedure lets the programmer identify the primary slot.

3.11.3.1 The Validation Procedure

For the example, we will construct a multi-direction web constraint that ties together three slots: ADDEND1, ADDEND2, and SUM. SUM is constrained to be ADDEND1 + ADDEND2, and ADDEND1 is constrained to be SUM - ADDEND2. ADDEND2 is not constrained. First, we will write the validation procedure:

```
void sum_two_slots_val (Am_Constraint_Context& cc, Am_Web_Events& events)
{
  events.End ();
  if (events.First ()) // No events
    return;
  Am_Slot slot = events.Get ();
  Am_Object self = slot.Get_Owner ();
  switch (slot.Get_Key ()) {
  case ADDEND1: { // ADDEND1 has changed last.
      int addend1 = self.GV (ADDEND1);
      int addend2 = self.GV (ADDEND2);
      self.SV (SUM, addend1 + addend2);
    }
    break;
  case ADDEND2: { // ADDEND2 has changed last.
      int prev_value = events.Get_Prev_Value ();
      int addend2 = self.GV (ADDEND2);
      events.Prev ();
      if (events.First () || events.Get ().Get_Key () != SUM) {
        int addend1 = self.GV (ADDEND1);
        self.SV (SUM, addend1 + addend2);
      }
     else { // SUM was set before ADDEND2. Must propagate previous result to ADDEND1.
        int sum = self.Get (SUM);
        self.SV (ADDEND1, sum - prev_value);
        self.SV (SUM, sum - prev_value + addend2);
      }
    }
    break;
  case SUM: { // SUM has changed last.
      int sum = self.GV (SUM);
      int addend2 = self.GV (ADDEND2);
      self.SV (ADDEND1, sum - addend2);
    }
    break;
  }
}
```

If the code looks complicated, it is because it is complicated. What makes writing web constraints difficult is that more than one slot can change between validations. The code above has to first check to see what slots have changed. Then it sees if something else has changed before that and then finally carries out the computation. To read which slots have changes, one uses the `Am_Web_Events` class. This contains a list of the slots that have changed value since the last time the web was validated in the order in which they were changed. This class is structured like the other iterator classes. It has `Start`, `End`, `Next`, `Prev`, `First`, and `Last` methods for traversing its items. Its `Get` method returns a `Am_Slot`. There is no `self` parameter because webs are not attached to a specific slot or object. A web could hypothetically float around between objects as it gets re-evaluated; though, this is generally not the case. To get a `self` pointer, one reads the owner of one of the slots in the event list. It should always be possible to determine a web's location using any one of its slots as a reference. The other method available in the events list is `Get_Prev_Value` which returns the value that the slot contained before it was changed. To get the slot's current value, one uses `Get` or `GV`.

Using `SV` in a web is similar to using `GV` except that it is used for output instead of input. Every slot set with `SV` will be assigned a constraint pointer to that web making the slot dependent on the web. It is legal to both `GV` and `SV` a single slot. In that case, the web would both be a constraint and a dependency on that slot. Both the constraint and dependency are kept whenever either are used in the validation procedure. Thus, in the above example, when `GV` is called on the `SUM` slot, the web's constraint on that slot will be kept. If neither `GV` or `SV` are called on a slot during validation, the dependency and/or constraint will be removed.

3.11.3.2 The Create and Initialization Procedures

The next procedure examined will be the create procedure. The semantics of the create procedure is that it returns true when it is passed the primary slot. For our example web, we have a choice of primary slots. It could be any of `ADDEND1`, `ADDEND2`, or `SUM`. We will choose `SUM`.

```
bool sum_two_slots_create (const Am_Slot& slot)
{
  return slot.Get_Key () == SUM;
}
```

Although our `sum_two_slots` web will normally be connected to three slots, when it is inherited, it is only connected to the primary slot. The purpose of the initialize procedure is to reconnect the web to all its other dependencies. Essentially, it is used to fix up the lost connections caused by inheritance.

```
void sum_two_slot_init (Am_Constraint_Context& /*cc*/, const Am_Slot& slot,
                        Am_Web_Init& init)
{
  Am_Object_Advanced self = slot.Get_Owner ();
  init.Note_Input (self, ADDEND1);
  init.Note_Input (self, ADDEND2);
  init.Note_Input (self, SUM);
  init.Note_Output (self, ADDEND1);
  init.Note_Output (self, ADDEND2);
  init.Note_Output (self, SUM);
}
```

The Am_Web_Init class is used to make dependency and constraint connections without performing a GV or SV on any slot. The method Note_Input creates a dependency and the method Note_Output creates a constraint. Some programmers may want to perform computation during initialization, in which case the usual cc parameter is available for calling GV or SV. The slot parameter provides the primary slot to which the web was attached.

3.11.3.3 Installing Into a Slot

A web is put into an object by setting it into its primary slot. First, a Am_Web variable is created using the three procedures defined above. This variable is stored into the primary slot just as a Am_Formula is stored, by calling Set.

```
Am_Web my_web (sum_two_slots_create, sum_two_slots_init,
               sum_two_slots_val);
object.Set (SUM, my_web);
```

Once a web is stored in the primary slot, its initialization procedure will take over and attach the web to any other slots it needs.

3.11.4 Using Am_Object_Advanced

There are several extra methods that can be used on any Amulet object that are not available in the regular Am_Object class. A programmer can manipulate these methods by typecasting a regular Am_Object into a Am_Object_Advanced class. For instance, in order to retrieve the Am_Slot form for a slot one uses Get_Slot:

```
#include OBJECT_ADVANCED__H  // Note need for special header file

Am_Object_Advanced obj_adv = (Am_Object_Advanced&)my_object;
Am_Slot slot = obj_adv.Get_Slot (Am_LEFT);
```

A programmer must be careful using the advanced object and slot classes. Many operations can break the object system if used improperly. A general principle should be to use the advanced features for a short time right after an object is first created. After the object is manipulated to add or change whatever is needed, the object should never need to be cast to advanced again.

A number of methods in the advanced object class are used to fetch slots. The method Get_Slot retrieves a slot and returns it as the advanced class Am_Slot. Get_Slot will always return a slot local to the object. If the slot was previously not local because it is still inherited or for other reasons, Get_Slot will make a placeholder slot locally in the object and return that. If the slot does not exist at all, Get_Slot will return Am_NULL_SLOT. There are two other methods used for fetching slots: Get_Owner_Slot, and Get_Part_Slot. These methods are similar to Get_Slot except they are to be used only for fetching part or owner slots. It is entirely possible to use Get_Slot instead of these specialized methods, but the specialized methods are more efficient. Other Am_Object_Advanced method are:

- Get_Slot_Locale (*slot_key*) - Returns the object where the given slot is locally defined. Note that being locally defined does not mean that the slot is not inherited for that object. Slots can become local when they are depended on by constraints for other similar reasons.

- `Disinherit_Slot (slot_key)` - Breaks the inheritance of a slot from the prototype object. This will delete formulas and values that were not set locally.
- `Print_Name_And_Data (ostream&)` - Used for debugging, this prints the contents of the object out in ASCII text.

3.11.5 Controlling Slot Inheritance

An innovation in Amulet is that the programmer can control the inheritance of each slot. This is useful if you want to make sure that certain slots are not shared by a prototype and its children. For example, the Amulet `Am_Window` object has a slot that points to the actual X/11 or MS Windows window object associated with that window. This slot should not be shared by multiple objects. The choices are defined by the enum `Am_Inherit_Rule` and are:

- `Am_INHERIT`: The default. Slots are inherited from prototypes to instances, and subsequent changes to the prototype will affect all instances that do not override the value.
- `Am_LOCAL`: The slot is never inherited by instances. Thus, the slot will not exist in an instance unless explicitly set there.
- `Am_COPY`: When the instance is created, a copy is made of the prototype's value and this copy is installed in the instance. Any further changes to the prototype will not affect the instance.
- `Am_STATIC`: All instances will share the same value. Changing the value in one object will affect all objects that share the slot. We have found this to be rarely useful.

To set the inheritance rule of the `Am_DRAWONABLE` slot of `new_win` to be local:

```
new_win.Set_Inherit_Rule (Am_DRAWONABLE, Am_LOCAL);
```

The inherit rule may also be set directly on a `Am_Slot`:

```
#include <am_inc.h> // defines OBJECT_ADVANCED__H for machine independance
#include OBJECT_ADVANCED__H // for slot_advanced

Am_Object_Advanced obj_adv = (Am_Object_Advanced&)new_win;
Am_Slot slot = obj_adv.Get_Slot(Am_DRAWONABLE);
slot.Set_Inherit_Rule(Am_LOCAL);
```

Using the `Am_Slot` method is useful when one wants to perform several manipulations on a single slot. By fetching the `Am_Slot` once and reusing the value, one can save the time needed to search for the slot.

The default rule with which an object will create all new slots added to an object can be changed using the `Set_Default_Inherit_Rule` method. Likewise, the current inherit rule can be examined using `Get_Default_Inherit_Rule`.

```
((Am_Object_Advanced&)my_object).Set_Default_Inherit_Rule (Am_COPY);
Am_Inherit_Rule rule = my_adv_object.Get_Default_Inherit_Rule ();
```

3.11.6 Controlling Formula Inheritance

For slots that are inherited normally, sometimes you still might want to control Formula inheritance separately. Remember from Section 3.7.4 that instances inherit formulas from their prototypes, but that setting the instance's slot normally removes the inherited formulas. There are times, however, when constraints from a prototype should be retained in instances *even if the instance's value is set.* For example, the `Am_VALUE` slot of widgets contain formulas that compute the value based on the user's actions. However, programmers are also allowed to set the `Am_VALUE` slot if they want the widget to reflect an application-computed value. In this case, the default formula in the `Am_VALUE` slot should *not* be removed if the programmer sets the slot. To achieve this, the programmer must set the slot's `Single_Constraint_Mode` to `false` (the default is `true`).

```
obj.Set_Single_Constraint_Mode (Am_VALUE, false);
```

The same parameter can be set directly on the slot as well:

```
Am_Object_Advanced obj_adv = (Am_Object_Advanced&)obj;
obj_adv.Get_Slot (Am_VALUE).Set_Single_Constraint_Mode (false);
```

Now, if `obj` contains a constraint, any instances of `obj` will always retain that constraint, even if another constraint or value is set into the instance. Furthermore, calls like `Remove_Constraint` on the instance's slot will still *not* remove the inherited constraint (though it will remove any additional constraints set directly into the instance).

3.11.7 Writing and Incorporating Demon Procedures

Amulet demons are special methods that are attached directly to an object or slot. Demons are used to cause frequently occurring, autonomous object behavior. The demons are written as procedures that are stored in an object's "demon set." The demon set is shared among objects that are inherited or copied from another.

The demon procedures that operate on an object have very specific purpose. There are five demons that can be overridden on the object level. Three of the demons deal with object creation and destruction, the other two handle part management.

Demons that are attached to slots behave similar to formulas. The slot demons are more generic than object level demons. Slot demons can detect when the slot value changes or is invalidated. Several slot demons can be assigned to a single slot making it possible for the slot to have multiple effects with a single event.

When a demon event occurs, the demon affected is put into the *demon queue* to be invoked later. All the demons put into demon queue are invoked, in order, whenever any slot is fetched by using `Get`. By invoking the demons on `Get`, Amulet can simulate the effects of eager evaluation because any demon that affects the value of different slots will be invoked whenever a slot is fetched.

3.11.7.1 Object Level Demons

The three demons that handle object creation and destruction are the create, copy, and destroy demons. Each demon is enqueued on its respective event. The create and copy demons get enqueued when the object is first created depending on whether the method `Create` or `Copy` was used to make the object. The destroy demon is never enqueued. Since the `Destroy` operation will cause the object to no longer exist, all demons that are already enqueued will be invoked and then the destroy demon will be called directly. This allows the programmer to still read the object while the destroy demon is running.

The creation/destruction demon procedures have the same parameter signature which takes the object affected by the demon. The type of the procedure is `Am_Object_Demon`.

```
// Here is an example create demon that initializes the slot MY_SLOT to
// be zero.
void my_create_demon (Am_Object self)
{
  self.Set (MY_SLOT, 0);
}
```

Two object-level demons are used to handle part-changing events. These are the add-part and the change-owner demons. The add-part and change-owner demons are always paired: the add-part demon for the part and the change-owner demon for the part's owner. Both demon procedures have the same parameter signature, three objects, which has the type `Am_Part_Demon`, but the semantics of each demon is different. The first parameter for both procedures is the `self` object -- the object being affected by the demon. The next two objects represent the change that has occurred. In the add-part demon, the second object parameter is an object that is being removed or replaced. The third parameter is an object that is being added or is replacing the object in the second parameter. For the change-owner demon, the semantics are reversed -- the second and third parameters represent the change that a part sees in its owner. The second parameter is the owner the part used to have, the third parameter is the new owner that has replaced the old owner.

```
// This owner demon checks to make sure that it's owner is a window. Any
// other owner will cause an error.
void my_owner_demon (Am_Object self, Am_Object prev_owner,
                     Am_Object new_owner)
{
  if (!new_owner.Is_Instance_Of (Am_Window))
    Am_Error ("You can only add me to a window!");
}
```

The events that generate add-part and change-owner demon events are methods such as `Add_Part`, `Remove_Part`, `Destroy`, and other methods that change the object hierarchy. Note that a given add-part demon always has a corresponding change-owner demon. The correspondence is not necessarily one to one because one can conceive of situations where one part is replacing another and thus two add-part calls can be associated with a single change-owner and vice versa.

Important note: The Opal and Interactor layers of Amulet define important demons for all of these object-level operations, so before setting a custom demon, the code should fetch the demon procedure currently stored in the demon set and call these in addition to the new demon. In a future release, we will make this more convenient to do.

```
void my_create_demon (Am_Object self)
{
  Am_Object_Demon_Type* proto_create = self.Get_Prototype ().
            Get_Demon_Set ().Get_Object_Demon (Am_CREATE_OBJ);
  if (proto_create)
      proto_create (self);  // Call prototype create demon.
  // Do my own code.
}
```

3.11.7.2 Slot Level Demons

Slot demons are not given permanent names like the object level demons. The slot demons are assigned a bit in a bit field to serve as their name. The slot demon procedure pointer is stored in the object. By turning on the bit in the slot, the slot will activate the demon procedure from the demon set whenever a triggering event occurs.

There are two parameters that control a slot demon. The first parameter distinguishes what event will trigger the demon. Slot demons can be triggered by one of two slot messages: the invalidate message or the value changed message. Most demons trigger on the value changed message because the demon's purpose is to note the change to other parts of the system. This can also be used to implement an active value scheme with a slot. Triggering using the invalidate message makes the demon act more like a formula. The demon can be used to revalidate the slot if desired. The *eager demon* uses this message to make Amulet formulas eager.

The other slot demon parameter is used to determine how often the demons will be triggered. Quite often, several slots affect the same demon in the same object. For instance, in a graphical object, the `Am_TOP` and `Am_LEFT` slots both affect the position of the object. A demon that handles object motion only needs to be triggered once if either of these slots changes. For this case, we use the per-object style. Whenever multiple slots change in a single object, the per-object demon will only be enqueued once. Only after the demon has been invoked will it reset and be allowed to trigger again. The other style of demon activation is per-slot. In this case, the demons act independently on each slot they are assigned. The demon triggers once for each slot and after it is invoked, it will be reset. The per-slot demon does not check to see if other demons have already been enqueued for the same object.

The slot demon procedure takes as its only parameter the slot that triggered the demon. If the demon could have been triggered by more than one slot (as can be the case when the demon is set to be per object), the slot provided is the very first one that triggered it.

```
// Here is an example slot demon. This demon does not do anything
// interesting, but it shows how the parameter can be used.
void my_slot_demon (Am_Slot slot)
{
  Am_Object_Advanced self = slot.Get_Owner ();
  self.Set (MY_SLOT, 0);
}
```

3.11.7.3 Modifying the Demon Set and Activating Slot Demons

To activate any demon, the object must know the demon procedure. Objects keep the list of available procedures in a structure called the *demon set* which is defined by the class `Am_Demon_Set`. Objects inherit their demon set from their prototype. The demon set is shared between objects in order to conserve memory. To modify the demon set of an object, one must first make the set a local copy. The demon set's `Copy` method is used to make new sets.

```
// Here we will modify the demon set of my_adv_obj by first making
// a copy of the old set and modifying it.  The new demon set is then
// placed back into the object.
Am_Demon_Set my_demons (my_adv_obj.Get_Demon_Set ().Copy ());
my_demons.Set_Object_Demon (Am_DESTROY_OBJ, my_destroy_demon);
my_adv_obj.Set_Demon_Set (my_demons);
```

When demon procedures are installed for object level demons, the demons will trigger on the next occurrence of their corresponding event. Note that the create and copy demon's events have already occurred for the prototype object where the demon procedures are installed. However, instances of the prototype as well as new copies will cause the new procedures to run. To make the demon procedure run for the current prototype object, one calls the demon procedure directly.

The demon set holds all demon procedures for the object, including the demons used in slots. The slot demons are installed by assigning each demon a bit name that will be stored in the slot. By setting the demon bit in a slot, events on that slot will activate the corresponding demon procedure. The bit name is represented by its integer value so bit 0 is number 1, bit 5 is number 32 (hex 0x0020). Section 3.11.7.5 discusses how to allocate demon bits.

```
// Here we install a slot demon that uses bit 5. The slot demon's semantics
// are to activate when the slot changes value and only once per object.
// Make sure that the demon set is local to the object (see above section).
my_demons.Set_Slot_Demon (0x0020, my_slot_demon,
                          Am_DEMON_ON_CHANGE | Am_DEMON_PER_OBJECT);
```

After the demon procedure is stored, one sets the bits on each slot that is able to activate the demon.

```
// Here we set a new bit to a slot. To make sure we do not turn off
// previously set bits, we first get the old bits and bitwise-or the new one.
Am_Slot slot = my_adv_obj.Get_Slot (MY_SLOT);
unsigned short prev_bits = slot.Get_Demon_Bits ();
slot.Set_Demon_Bits (0x0020 | prev_bits);
```

To cause newly created objects to have certain demon bits set, one changes the default demon bits.

```
// Make the new slot demon default.
unsigned short default_bits = my_adv_obj.Get_Default_Demon_Bits ();
default_bits |= 0x0020;
my_adv_obj.Set_Default_Demon_Bits (default_bits);
```

Another factor in slot demon maintenance is the *demon mask*. The demon mask is used to control whether the presence of a demon bit in a slot will force the slot to make a temporary slot in every instance. A temporary slot is used by ORE to provide a local slot in an object even when the value of the slot is inherited. If a temporary slot is not available, then there will be no demon run for that object. This is necessary when one wants inherited objects to follow the behavior of a prototype

object. For instance, in a rectangle object, if one changes the Am_LEFT slot in the prototype, one would like a demon to be fired for the Am_LEFT slot in every instance. That requires there to be a temporary slot for every instance. Set the demon mask bit for all demons that require a temporary slot. For all other demons put zero. A temporary slot will be created for slots whose demon bits contain at least one bit stored in the demon mask.

```
// Setting the demon mask
unsigned short mask = my_adv_obj.Get_Demon_Mask ();
mask |= 0x0020;  // add the new demon bit.
my_adv_obj.Set_Demon_Mask (mask);
```

3.11.7.4 The Demon Queue

The demon queue is where demon procedures are stored when their events occur. Objects hold the demon queue in the same way that they keep their demon set: the same queue is shared when objects are instanced or copied. However, Amulet never uses more than one demon queue. There is only one global queue for all objects. It is possible to make a new queue and store it in an object, but it never happens.

```
// Here is how to make a new queue.
// It is unlikely that anyone will need to do this.
Am_Demon_Queue my_queue;
my_adv_obj.Set_Queue (my_queue);
```

To find and manipulate the global demon queue, one can take any object and read its queue.

```
Am_Demon_Queue global_queue =
          ((Am_Object_Advanced&)Am_Root_Object).Get_Queue ();
```

The demon queue has two basic operations: enqueuing a new demon into the list and causing the queue to invoke. Invoking the queue causes all stored demon procedures to be read out of the queue, in order, and executed. While the queue is invoking, it cannot be invoked recursively. This prevents the queue from being read out of order while a demon is still running.

The demon queue is automatically invoked in some circumstances. First, the queue is invoked whenever the method, Get, is called on an object. This makes sure that any demons that affect the slot being retrieved are brought up to date. Another time is when the Destroy method is called on the object. The other time the queue is invoked is when updating occurs in the main loop and other window updating procedures. When windows are updated, the demon queue is invoked to update changed object values.

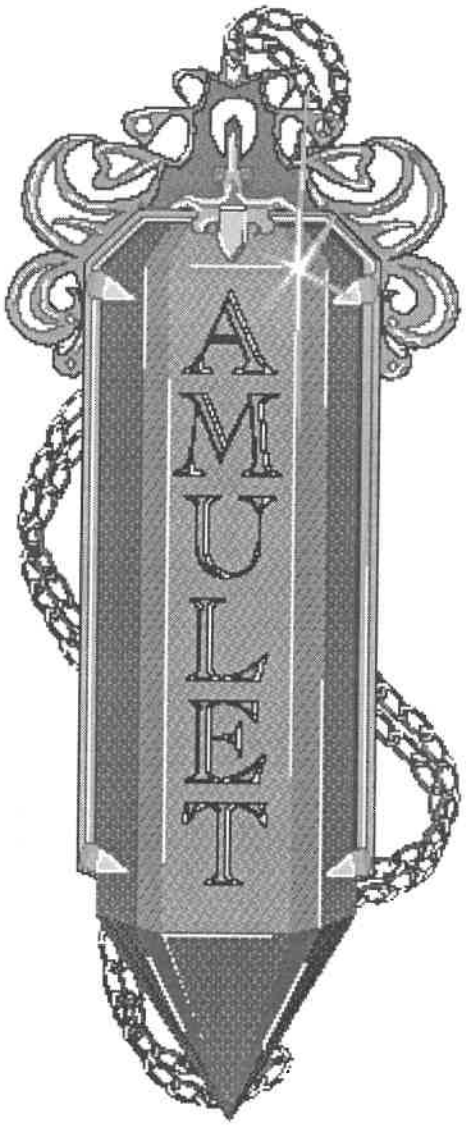
The demon queue is not a true queue in that it does not have a dequeue operation. The dequeue is wrapped in the Invoke method. The queue does have a means for deleting entries. One deletes all the demons that have a given slot or object as a parameter by using the Delete method.

3.11.7.5 How to Allocate Demon Bits and the Eager Demon

In order to develop new slot demons, one must provide a bit name for it. Presently, Amulet does not provide a means for dispensing bit names for demons. To see if a demon bit is being used by an object, read the slot demons from the demon set and see which bits are not being used. This procedure is presently sufficient since one never modifies an object's demon set more than once. Generally, only prototype objects need to be manipulated and one can often know which demons are set in a given prototype object.

Some of the demon bits are off limits to Amulet programmers. Amulet reserves two bits for use by the object system and another three bits for opal. The object system uses bits 0 and 1, opal uses bits 2, 3, and 4. Bits 5, 6, and 7 are available for programmers. Presently there are only the eight bits available for slot demons.

Bit 0 in the object system is for the *eager demon*. The eager demon is a default demon that all slots use. The demon is used to validate the slot whenever it becomes invalid. This makes the formula validation scheme eager hence the name. A programmer can turn off eager evaluation by turning off the eager bit in all the slots that one wants to be lazy. One can also set the eager demon procedure to be NULL in the demon set. When adding new demons to a slot, one must be careful not to turn off the eager bit by accident.



4. Opal Graphics System

This chapter describes simple graphical objects, styles, and fonts in Amulet. "Opal" stands for the Object Programming Aggregate Layer. Opal makes it easy to create and manipulate graphical objects. In particular, Opal automatically handles object redrawing when properties of objects are changed.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

4.1 Overview

This chapter describes the Opal graphical object system. The text assumes that the reader is familiar with Amulet Objects, Slots, and Constraints, as presented in the Amulet Tutorial and the ORE chapter.

4.1.1 Include Files

There are several data types documented in this chapter, declared in several different header files. You only need to include `amulet.h` at the top of your files, but some programmers like to look at header files. Here is a list of most of the objects and other data types discussed in this chapter, along with the header file in which they are declared.

- `gdefs.h`: `Am_Style`, `Am_Font`, `Am_Point_List`, `Am_Image_Array`
- `opal.h`: default `Am_Style`'s, default `Am_Font`'s, `Am_Screen`, `Am_Graphical_Object`, `Am_Window`, `Am_Rectangle`, `Am_Roundtangle`, `Am_Line`, `Am_Arrow`, `Am_Polygon`, `Am_Arc`, `Am_Text`, `Am_Bitmap`, `Am_Group`, `Am_Map`, default constraints, `Am_Initialize`, `Am_Cleanup`, `Am_Beep`, `Am_Move_Object`, `Am_To_Top`, `Am_To_Bottom`, `Am_Create_Screen`, `Am_Update`, `Am_Update_All`, `Am_Do_Events`, `Am_Main_Event_Loop`, `Am_Exit_Main_Event_Loop`, default `Am_Point_In_` functions, `Am_Translate_Coordinates`, `Am_Merge_Pathname`
- `value_list.h`: `Am_Value_List`
- `text_fns.h`: all text editing functions, `Am_Edit_Translation_Table`

For more information on Amulet header files and how to use them, see Section 1.6 in the Overview chapter.

4.2 The Opal Layer of Amulet

Opal, which stands for the Object Programming Aggregate Layer, provides simple graphical objects for use in the Amulet environment. The goal of Opal is to make it easy to create and edit graphical objects. To this end, Opal provides default values for all of the properties of objects, so simple objects can be drawn by setting only a few parameters. If an object is changed, Opal automatically handles refreshing the screen and redrawing that object and any other objects that may overlap it. Objects in Opal can be connected together using *constraints*, which are relations among objects that are declared once and automatically maintained by the system. An example of a constraint is that a line must stay attached to a rectangle. Constraints are discussed in the Tutorial and the ORE chapter.

Opal is built on top of the Gem module, which is the Graphics and Events Module that refers to machine-specific functions. Gem provides an interface to both X windows and to Windows NT, so applications implemented with Opal objects and functions will run on either platform without modification. Gem is described in chapter 7.

To use Opal, the programmer should be familiar with Amulet objects and constraints as presented in the Tutorial. Opal is part of the Amulet library, so all objects discussed in this chapter are accessible by linking the Amulet library to your program (see the Overview for instructions). Opal will work with any window manager on top of X/11, such as mwm, uwm, twm, etc. Opal provides support for color and gray-scale displays.

Within the Amulet environment, Opal forms an intermediary layer. It uses facilities provided by the ORE object and constraint system, and provides graphical objects that can be combined to build more complicated gadgets. Opal does not handle any input from the keyboard or mouse -- that is handled by the separate *Interactors* module, which is described in chapter 5. The Amulet Widgets, such as scroll-bars and menus, are partially built out of Opal objects and partly by calling Gem directly (for efficiency). Widgets are generally more complicated than the Opal objects, usually consisting of interactors attached to graphics, and are discussed in chapter 6.

4.3 Basic Concepts

4.3.1 Windows, Objects, and Groups

X/11, Windows NT, and Mac Quickdraw all allow you to create windows on the screen. In X they are called "drawables", and in Windows NT and on the Mac they are just called "windows". An Opal *window* is an ORE data structure that contains pointers to these machine-specific structures. Opal windows can be nested inside other windows to form *subwindows*. Windows clip all graphics so they do not extend outside the window's borders.

Each window defines a new coordinate system with (0,0) in the upper-left corner. The coordinate system uses the pixel as its unit of measurement. Amulet does not support other coordinate spaces, such as inches or centimeters, and it does not support transformations that take the display's aspect ratio into account. Amulet windows are discussed fully in Section 4.10.

The basics of Object Oriented Programming are beyond the scope of this chapter. The objects in Opal use the ORE object system, and therefore operate as a prototype-instance model. This means that each object can serve as a prototype (like a class) for any further instances; there is no distinction between classes and instances. Each graphic primitive in Opal is implemented as an object. When the programmer wants to cause something to be displayed in Opal, it is necessary to create instances of these graphical objects. Each instance remembers its properties so it can be redrawn automatically if the window needs to be refreshed or if objects change.

A *group* is a special kind of object that holds a collection of other objects. Groups can hold any kind of graphic object including other groups, but an object can only be in one group at a time. Groups form a pure hierarchy. The objects that are in a group are called *parts* of that group, and the group is called the *owner* of each of the parts. Groups, like windows, clip their contents to the bounding box defined by their left, top, width, and height. Groups also define their own coordinate system, so that the positions of their parts are relative to the left and top of the group.

The graphical hierarchy in Amulet has `Am_Screen`, the display device, at its root. Am_Screen's graphical parts are windows. Those windows can contain other windows, groups, or graphical objects as parts. No object is drawn unless `Am_Screen` is one of its ancestors in the graphical hierarchy. It is important to remember to add all windows to `Am_Screen`, or to another window on the screen, and to add all groups and objects to a window. If you do not, nothing will be displayed.

Non-graphical objects, like Interactors and Command objects (and application specific objects) can be added as parts to any kind of object, but graphical objects will only be displayed if they are added as parts to an `Am_Window` or a `Am_Group` (or an instance of thse objects).

4.3.2 The "Hello World" Example

An important goal of Opal is to make it simple to create pictures, hiding most of the complexity of the machine dependant graphics toolkits. We provide default values for all of the properties of graphical objects in Amulet, which makes it easy to use these objects in the most common way, while still providing the option of complete customization for more specialized applications.

A traditional introductory program is called Hello World. This program displays the string "Hello world" in a window on the screen, and automatically refreshes the window if it is obscured. In Amulet, Hello World is a very short program:

```
#include <amulet.h>

main (void)
{
  Am_Initialize ();

  Am_Screen
    .Add_Part (Am_Window.Create ("window")
      .Set (Am_WIDTH, 200)
      .Set (Am_HEIGHT, 50)
      .Add_Part (Am_Text.Create ("string")
        .Set (Am_TEXT, "Hello World!")));

  Am_Main_Event_Loop ();
  Am_Cleanup ();
}
```

This code is provided in the Amulet source code in the directory `samples/hello/`. The same source code is used on all three platforms: Unix/X, Windows NT/'95, and Mac.

Note that the Amulet code has a declarative programming feel to it, instead of standard imperative programming. Instead of giving Amulet step by step instructions on how to draw things, such as, "first draw a window, then draw a string," and so on, you tell Amulet, "Create a window. Create some text, and add it to the window. Go!" The programmer never calls "draw" or "erase" methods on objects. Once you set up the graphical world how you want it, and "Go" (start the main event loop), Opal automatically draws and erases the the right things at the right time.

Section 4.5 presents all the kinds of objects available in Opal.

4.3.3 Initialization and Cleanup

Amulet requires a call to `Am_Initialize()` before referencing any Opal objects or classes. This function creates the Opal prototypes and sets up bookkeeping information in Amulet objects and classes. Similarly, a call to `Am_Cleanup()` at the end of your program allows Amulet to destroy the prototypes and classes, explicitly releasing memory and graphical resources that might otherwise remain occupied after the program exits.

4.3.4 The Main Event Loop

In order for interactors to perceive input from the mouse and keyboard, the main-event-loop must be running. This loop constantly checks to see if there is an event, and processes it if there is one. The automatic redrawing of graphics also relies on the main-event-loop. Exposure events, which occur when one window is uncovered or *exposed*, cause Amulet to refresh the window by redrawing the objects in the exposed area.

A call to `Am_Main_Event_Loop()` should be the second-to-last instruction in your program, just before `Am_Cleanup()`. Your program will continue to run until you tell it otherwise. All Amulet programs running `Am_Main_Event_Loop()` can be aborted by pressing the escape sequence, which by default is `META_SHIFT_F1`. Most programs will have a Quit option, in the form of a button or menu item, that calls the `Am_Exit_Main_Event_Loop()` routine, which will cause the main event loop to terminate.

4.3.5 Am_Do_Events

Amulet does not yet have support for animation, or other mechanisms for connecting to outside applications and databases. To implement these using Amulet, you have to write your own event loop (instead of using `Am_Main_Event_Loop`) and perform the external or time-based actions yourself. This section explains how to write your own loop.

Normally, in response to an input event, applications will make some changes to graphics or their internal state and then return. Sometimes, however, an application might want to make a graphical change, have it seen by the user while waiting a little, and then make another change. This might be necessary to make something blink a few times, or for a short animation. To do this, an application must call `Am_Do_Events()` to cause Amulet to update the screen based on all the changes that have happened so far. Eventually, Amulet will contain an Animation Interactor, but it is not implemented yet.

You might also use `Am_Do_Events()` to create your own event loop when Amulet's default event loop is not adequate. This might happen if you use Amulet with some other toolkit which also has its own main event loo. You might need to monitor non-Amulet queues, processes or inter-process-communication sockets. Calling `Am_Do_Events()` repeatedly in a loop will cause all the Interactors and Amulet activities to operate correctly, because the standard `Am_Main_Event_Loop` essentially does the same thing as calling `Am_Do_Events()` in a loop. `Am_Do_Events()` never

blocks waiting for an event, but returns immediately whether there is anything to do or not. The return boolean from Am_Do_Events tells whether the whole application should quit or not. A main-loop might look like:

```
main (void) {
  Am_Initialize ();
  ... // do any necessary set up and object creation

  // use the following instead of Am_Main_Event_Loop ();
  bool continue_looping = true;
  while (continue_looping) {
     continue_looping = Am_Do_Events();
     ... // check other queues or whatever
  }
  Am_Cleanup ();
}
```

4.4 Slots Common to All Graphical Objects

4.4.1 Left, Top, Width, and Height

All graphical objects have Am_LEFT, Am_TOP, Am_WIDTH, and Am_HEIGHT slots that determine their position and dimensions. Some objects have simple numerical values in these slots, and some have formulas that compute these values based on other properties of the object. Check the section below for a specific object to find its default values for these slots. All values must be ints.

For all graphical objects, Opal never draws outside the bounding box specified by the Am_LEFT, Am_TOP, Am_WIDTH, and Am_HEIGHT of the object.

4.4.2 Am_VISIBLE

The Am_VISIBLE slot of all windows and graphical objects contains a bool value which determines whether the object is visible or not. Objects that are not visible are not drawn on the screen. In a window, Am_VISIBLE controls whether the window is drawn on the screen. To make a group and all of its parts invisible, it is sufficient to set the Am_VISIBLE slot of the group to false.

Invisible objects are typically ignored by interactors and graphics routines. For example, you cannot use interactors to select an invisible object with the mouse, even if you click on the area where the invisible object would appear. Also, invisible parts of a group are generally not taken into account when the size of the group is computed.

4.4.3 Line Style and Filling Style

The Am_LINE_STYLE and Am_FILL_STYLE slots hold instances of the Am_Style class. If an object has a style in its Am_LINE_STYLE slot, it will have a border of that color. If it has a style in the Am_FILL_STYLE slot, it will be filled with that color. Other properties such as line thickness and stipple patterns are determined by the styles in these slots.

Often you do not have to create customized instances of Am_Style to change the color of an object. You can use a predefined style such as Am_Red instead. Styles are fully documented in Section 4.6.

Using special value Am_No_Style or NULL in the Am_LINE_STYLE or Am_FILL_STYLE slot will cause the object to have no border or no fill.

4.4.4 Am_HIT_THRESHOLD and Am_PRETEND_TO_BE_LEAF

The Am_HIT_THRESHOLD, Am_PRETEND_TO_BE_LEAF, and Am_VISIBLE slots are used by functions which search for objects given a rectangular region or an (x,y) coordinate. For example, suppose a mouse click in a window should select an object from a group of objects. When the mouse is clicked, Amulet compares the location of the mouse click with the size and position of all the objects in the window to see which one was selected.

First of all, only visible objects can be selected this way. If an object's Am_VISIBLE slot contains false, it will not respond to events such as mouse clicks with conventional Interactors programming techniques.

The Am_HIT_THRESHOLD slot controls the sensitivity of functions that decide whether an event (like a mouse click) occurred "inside" an object. If the Am_HIT_THRESHOLD of an object is 3, then an event 3 pixels away from the object will still be interpreted as being "inside" the object. The default value of Am_HIT_THRESHOLD for all Opal objects is 0. **Note:** it is often necessary to set the Am_HIT_THRESHOLD slot of all groups that are owners of the target object; if an event occurs "outside" of a group, then the selection functions will not check the parts of the group.

When the value of a group's Am_PRETEND_TO_BE_LEAF slot is true, then the selection functions will treat that group as a leaf object (even though the group has parts). See Section 4.9.2 regarding the function Am_Point_In_Leaf. Also, consult the Interactors chapter regarding the function Am_Inter_In_Leaf.

4.5 Specific Graphical Objects

The descriptions in this section highlight aspects of each object that differentiate it from other objects. Some properties of Opal objects are similar for all objects, and are documented in Section 4.4. All of the exported objects in Amulet are summarized in chapter 10.

4.5.1 Am_Rectangle

Am_Rectangle is a rectangular shaped object with a border of Am_LINE_STYLE and filled with Am_FILL_STYLE. The default rectangle is a 10 by 10 pixel square located at (0, 0), drawn with black line and fill styles.

Slot	Default Value	Type	
Am_VISIBLE	true	bool	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	10	int	
Am_HEIGHT	10	int	
Am_FILL_STYLE	Am_Black	Am_Style	*Inside of rectangle*
Am_LINE_STYLE	Am_Black	Am_Style	*Edge of rectangle*

4.5.2 Am_Line

Am_Line is a single line segment with endpoints (Am_X1, Am_Y1) and (Am_X2, Am_Y2) drawn in Am_LINE_STYLE (Am_FILL_STYLE is ignored). Am_X1, Am_Y1, Am_X2, Am_Y2, Am_LEFT, Am_TOP, Am_WIDTH, and Am_HEIGHT are constrained in such a way that if any one of them changes, the rest will automatically be updated so the endpoints of the line and its bounding box are always consistent.

Slot	Default Value	Type	
Am_LINE_STYLE	Am_Black	Am_Style	
Am_X1	0	int	*Am_X1, Am_Y1, Am_X2, Am_Y2, Am_LEFT, Am_TOP, Am_WIDTH, and Am_HEIGHT are constrained in such a way that if any one of them changes, the rest will automatically be updated to reflect that change.*
Am_Y1	0	int	
Am_X2	0	int	
Am_Y2	0	int	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	1	int	
Am_HEIGHT	1	int	
Am_VISIBLE	true	bool	
Am_HIT_THRESHOLD	0	int	

4.5.3 Am_Arc

Am_Arc is used to draw circles, ellipses, and arc and pie shaped segments of circles and ellipses. Am_ANGLE1 determines the origin of the segment, measured in degrees counterclockwise from 3 o'clock. Am_ANGLE2 specifies the end of the arc segment, measured in degrees counterclockwise from Am_ANGLE1. Some examples: To draw a circle, Am_ANGLE1 can have any value, but Am_ANGLE2 must be 360 degrees or greater. To draw a semicircle from 12 o'clock counterclockwise to 6 o'clock, Am_ANGLE1 would be 90, and Am_ANGLE2 would be 180.

Arcs are filled as pie pieces to the center of the oval when a colored filling style is provided.

Slot	Default Value	Type	
Am_VISIBLE	true	bool	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	10	int	
Am_HEIGHT	10	int	
Am_ANGLE1	0	0..360	*Origin, degrees from 3 o'clock*
Am_ANGLE2	360	0..360	*Terminus, distance from origin*
Am_FILL_STYLE	Am_Black	Am_Style	*Inside of arc*
Am_LINE_STYLE	Am_Black	Am_Style	*Edge of arc*

4.5.4 Am_Roundtangle

Instances of the Am_Roundtangle prototype are rectangles with rounded corners.

Slot	Default Value	Type	
Am_VISIBLE	true	bool	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	10	int	
Am_HEIGHT	10	int	
Am_RADIUS	Am_SMALL_RADIUS	Am_Radius_Flag or int	*{ Am_SMALL_RADIUS, Am_MEDIUM_RADIUS, Am_LARGE_RADIUS }*
Am_FILL_STYLE	Am_Black	Am_Style	*Inside of roundtangle*
Am_LINE_STYLE	Am_Black	Am_Style	*Edge of roundtangle*

The slots in this object are the same as Am_Rectangle, with the additional slot Am_RADIUS, which specifies the curvature of the corners. The value of the Am_RADIUS slot can either be an integer, indicating an absolute pixel radius for the corners, or an element of the enumerated type Am_Radius_Flag, indicating a small, medium, or large radius (see table below). The keyword values do not correspond directly to pixel values, but rather compute a pixel value as a fraction of the length of the shortest side of the bounding box.

Value of Am_RADIUS	Fraction
Am_SMALL_RADIUS	1/5
Am_MEDIUM_RADIUS	1/4
Am_LARGE_RADIUS	1/3

Figure 4-1 shows the meanings of the slots of Am_Roundtangle. If the value of Am_RADIUS is 0, the roundtangle looks just like a rectangle. If the value of Am_RADIUS is more than half the shortest side (which would mean there is not room to draw a corner of that size), then the corners are drawn as large as possible, as if the value of Am_RADIUS were half the shortest side.

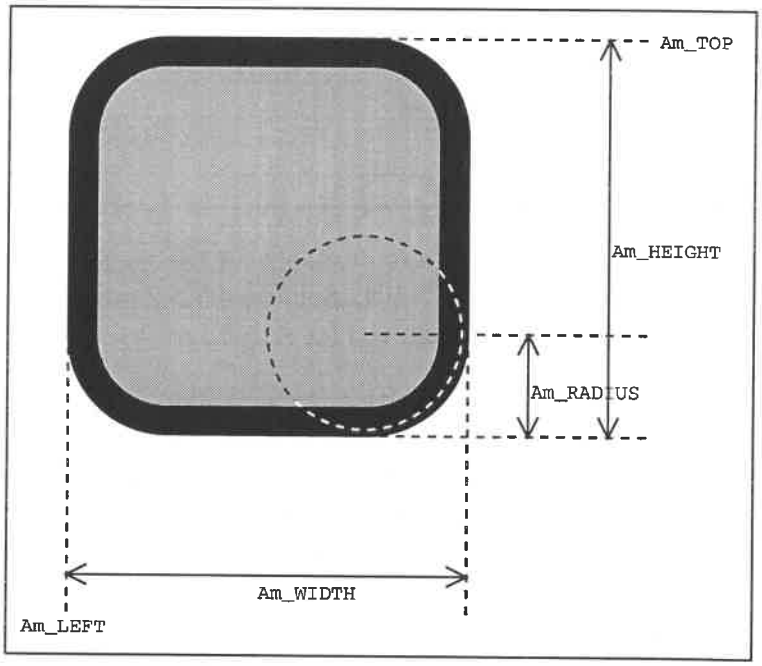


Figure 4-1: The parameters of a roundtangle.

4.5.5 Am_Polygon

The interface to the Am_Polygon object is more complicated than other Opal objects. To specify the set of points that should be drawn for the polygon, you must first create an instance of Am_Point_List with all your (x,y) coordinates, and then install the point list in the Am_POINT_LIST slot of your Am_Polygon object.

Slot	Default Value	Type
Am_VISIBLE	true	bool
Am_LEFT	<formula>	int
Am_TOP	<formula>	int
Am_WIDTH	<formula>	int
Am_HEIGHT	<formula>	int
Am_POINT_LIST	empty Am_Point_List	Am_Point_List
Am_LINE_STYLE	Am_Black	Am_Style
Am_FILL_STYLE	Am_Black	Am_Style
Am_HIT_THRESHOLD	0	int
Am_WANT_PENDING_DELETE	false	bool
Am_SELECT_OUTLINE_ONLY	0	bool
Am_SELECT_FULL_INTERIOR	0	bool

Section 4.5.5.1 lists all of the functions that are available for the Am_Point_List class, including how to create point lists and add points to the list. Section 4.5.5.2 provides an example of how to create a polygon using the Am_Polygon object and the Am_Point_List class.

A polygon's point list (Am_POINT_LIST) and bounding box (Am_LEFT, Am_TOP, Am_WIDTH, and Am_HEIGHT) are related by a web constraint that reevaluates whenever one of these slots changes. Whenever one of the bounding box slots is changed, all the points in the point list are translated or scaled so that the resulting polygon has that bounding box. The Am_LINE_STYLE slot is also involved in this constraint, since the polygon's bounding box may change when the thickness or cap style of its border changes.

Conversely, whenever a new point list is installed in the Am_POINT_LIST slot, the bounding box slots are recalculated. If you make a destructive modification to the point list class currently installed in the slot, such as adding a point to it with Add(), then you must call Note_Changed() on the polygon's Am_POINT_LIST slot to reevaluate and redraw the polygon. (For more details about destructive modification of slot values and formula evaluation, see Section 3.11.1.)

Because of the web in the polygons' bounding box and point list slots, the effects of certain Set operations will change depending on the order of the Sets. If you set the Am_POINT_LIST slot of a polygon, and then set its Am_WIDTH and Am_HEIGHT, the polygon will first get the correct point list, and then be scaled to the correct size. If you set the size first, however, that change will be masked by the calculate size of the point list. You should always make changes to the size and location of a polygon **after** you set its point list, to make sure those changes are actually seen.

The shape defined by an Am_Polygon object does not have to be a closed polygon. To draw a closed polygon, make the first and last points in the Am_Point_List the same. If they are not the same, and a fill style is provided, the border of the fill style will be along an invisible line between the first and last points.

Which sections of a self-intersecting polygon (such as a 5-pointed star) are displayed is controlled by the Am_Fill_Poly_Flag part of the Am_FILL_STYLE used for the polygon. See Section 4.6.3.6.

4.5.5.1 The Am_Point_List Class

Am_Point_List is a list of x-y coordinates, stored as pairs of floats, in the coordinate system of the polygon's owner. It is implemented as a wrapper class with an interface very similar to the standard Am_Value_List class (Section 3.8). Each Am_Point_List object contains a current-element pointer which can be moved forward and backward in the list. The current point is used for retrieving, replacing, inserting, and deleting points in the list.

In addition to the standard list-management functions, Am_Point_List also provides methods for finding the points' bounding box, translating all the points by an x-y offset, and scaling all the points relative to an origin.

The Am_Point_List is designed to be used to specify the vertices of a polygon object. Section 4.5.5.2 contains an example of how to use the Am_Point_List class with the Am_Polygon object

These methods are constructors for Am_Point_List:

- Am_Point_List() Return an empty list. (This constructor is implicitly called when you declare a variable of type Am_Point_List.)
- Am_Point_List(ar, size) Return a list initialized from ar, which is a flat array of values { x1 y1 x2 y2 ... xn yn }, where size is 2n, and the points may be either ints or floats.

These methods add new points to the list. In each case, if destructive modification is desired, pass false as an additional, final argument.

- Add(x, y) Adds a new point (x, y) to the tail of the list. Returns a reference to the list, so multiple Add() calls can be chained together like "list.Add(x,y).Add(x,y)..."
- Add(x, y, Am_HEAD) Adds a new point (x, y) to the head of the list. Returns a reference to the list.
- Append(other_list) Appends the points from another Am_Point_List to the end of the list. Returns a reference to the list.

These methods move the current point:

- Start() Makes the first point current. (Legal even if the list is empty.)
- End() Makes the last point current. (Legal even if the list is empty.)
- Prev() Makes the previous point current, wrapping around if current is first point of list.

- `Next()` Makes the next point current, wrapping around if current is last point of list.
- `First()` Returns true when current passes first point of list.
- `Last()` Returns true when current passes last point of list.

These methods get the current point and other list information:

- `Get(int &x, int &y)` Sets x and y to the (rounded) x-y coordinates of current point. Error if no point is current.
- `Get(float &x, float &y)` Sets x and y to the x-y coordinates of current point. Error if no point is current.
- `Length()` Returns length of the list.
- `Empty()` Returns whether list is empty.

These methods modify the list at the current point. In each case, if destructive modification is desired, pass false as an additional, final argument.

- `Set(x, y)` Sets the current point to (x,y). Error if no point is current.
- `Insert(x, y, Am_BEFORE)` Inserts (x,y) before the current point.
- `Insert(x, y, Am_AFTER)` Inserts (x,y) after the current point
- `Delete()` Deletes the current point.

These methods operate on the list as a set of points. For `Translate()` and `Scale()`, if destructive modification is desired, pass false as an additional, final argument.

- `Get_Extents(int &min_x, int &min_y, int &max_x, int &max_y)` Sets its arguments to the coordinates of the smallest rectangle (with integral coordinates) that completely encloses the points in the list.
- `Translate(offset_x, offset_y)` Transforms every point in the list by adding `offset_x` to its x coordinate and `offset_y` to its y coordinate.
- `Scale(scale_x, scale_y, origin_x, origin_y)` Transforms every point in the list by scaling its vector from (`origin_x`, `origin_y`) by `scale_x` in the x direction and `scale_y` in the y direction.

The member functions of the `Am_Point_List` class are invoked using the standard C++ dot notation, as in "`my_point_list.Add (10, 20);`".

4.5.5.2 Using Point Lists with Am_Polygon

The list of points for a polygon should be installed in an instance of `Am_Point_List`, and then that point list should be set into the `Am_POINT_LIST` slot of your `Am_Polygon` object. The constructors for `Am_Point_List` allow you to initialize your point list with some, all, or none of the points that you will eventually use. After the point list has been created, you can add and remove points from it.

Here is an example of a triangle generated by adding points to an empty point list. The point list is then installed in an Am_Polygon object. To see the graphical result of this example, add the triangle_polygon object to a window.

```
Am_Point_List triangle_pl;
triangle_pl
  .Add (15, 50)
  .Add (45, 10)
  .Add (75, 50);

Am_Object triangle_polygon = Am_Polygon.Create ("triangle_polygon")
  .Set (Am_POINT_LIST, triangle_pl)
  .Set (Am_LINE_STYLE, Am_Line_2)
  .Set (Am_FILL_STYLE, Am_Yellow);
```

Here is an example of a five-sided star generated from an array of integers. To see the graphical result of this example, add the star_polygon object to a window.

```
static int star_ar[12] = {100, 0, 41, 181, 195, 69, 5, 69, 159, 181,
                          100, 0};

Am_Point_List star_pl (star_ar, 12);

Am_Object star_polygon = Am_Polygon.Create ("star_polygon")
  .Set (Am_POINT_LIST, star_pl)
  .Set (Am_FILL_STYLE, Am_No_Style);
```

4.5.6 Am_Text

The Am_Text object is used to display a single line of text in a specified font. The Am_TEXT slot holds the string to be displayed, and the Am_FONT slot holds an instance of Am_Font.

Slot	Default Value	Type	
Am_VISIBLE	true	bool	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	<formula>	int	
Am_HEIGHT	<formula>	int	
Am_TEXT	""	Am_String	*String to display*
Am_FONT	Am_Default_Font	Am_Font	
Am_CURSOR_INDEX	Am_NO_CURSOR	int	*Position of cursor in string*
Am_LINE_STYLE	Am_Line_2	Am_Style	*Color of text, cursor*
Am_FILL_STYLE	Am_No_Style	Am_Style	*Background behind text*
Am_X_OFFSET	<formula>	int	
Am_INVERT	false	bool	*Exchanges line and fill style*

The Am_CURSOR_INDEX slot determines where a text insertion cursor (a vertical line) will be drawn. The slot contains an integer specifying the position of the cursor, measured in number of characters from the start of the string. A value of zero places the cursor before the text, and a value of Am_NO_CURSOR turns off the cursor.

The Am_WIDTH and Am_HEIGHT slots contain formulas that calculate the size of the object according to the string and the font being displayed. If the value of either of these slots is set, the formula will be removed, and the text object will no longer change size if the string or font is changed. In that case, a formula in the Am_X_OFFSET slot scrolls the text left and right to make sure the cursor is always visible.

The Am_LINE_STYLE slot controls the color of the text and the thickness of the cursor. If a style is provided in the Am_FILL_STYLE slot, then the background behind the text will be filled with that color, otherwise the background is transparent. Setting Am_INVERT to true causes the line and filling styles to be switched, which is useful for "highlighting" the text object. If Am_INVERT is true but no fill style is provided, Amulet draws the text as white against a background of the line style color.

4.5.6.1 Fonts

Am_Font is a C++ class defined in gdefs.h. Its creator functions are used to make customized instances describing the desired font. You can create fonts either with standard parameters that are more likely to be portable across different platforms, or by specifying the name of a specific font. The properties of fonts are: *family* (fixed, serif, or sans-serif), *face* (bold, italic, and/or underlined), and *size*. Am_Default_Font, exported from opal.h, is the fixed-width, medium-sized font you would get from calling the Am_Font constructor with its default values. Allowed values of the standard parameters appear below.

In the creator functions for Am_Font, the allowed values for the *family* parameter are:

- Am_FONT_FIXED a fixed-width font, such as Courier.
- Am_FONT_SERIF a variable-width font with "serifs", such as Times.
- Am_FONT_SANS_SERIF a variable-width font with no serifs, such as Helvetica.

The allowed values for the *size* parameter are:

- Am_FONT_SMALL a small size: about 10 pixels tall
- Am_FONT_MEDIUM a normal size: about 12 pixels tall
- Am_FONT_LARGE a large size: about 18 pixels tall
- Am_FONT_VERY_LARGE a larger size: about 24 pixels tall

4.5.6.2 Functions on Text and Fonts

There are additional functions that operate on Am_Text objects, strings, and fonts declared in the header file text_fns.h. These functions are included in the standard Amulet library, but are not automatically included by amulet.h because of their infrequent use. The Am_Text_Interactor uses these functions to edit strings. To access these functions directly, add the following lines to the top of your Amulet program:

```
#include <am_inc.h> // defines TEXT_FNS__H for machine independance
#include TEXT_FNS__H
```

4.5.6.3 Editing Text

Text editing is a feature provided by the Interactors module. To make a text object respond to mouse clicks and the keyboard, you need to use an `Am_Text_Interactor`. For details on this and other Interactors, see chapter 5.

4.5.7 Am_Bitmap

To specify the image that should be drawn by the `Am_Bitmap` object, you must first create an instance of `Am_Image_Array` containing the image data, and then install the image in the `Am_IMAGE` slot of your `Am_Bitmap` object.

Slot	Default Value	Type	
`Am_VISIBLE`	`true`	`bool`	
`Am_LEFT`	`0`	`int`	
`Am_TOP`	`0`	`int`	
`Am_WIDTH`	`<formula>`	`int`	
`Am_HEIGHT`	`<formula>`	`int`	
`Am_LINE_STYLE`	`Am_Black`	`Am_Style`	*Color of on pixels*
`Am_FILL_STYLE`	`Am_No_Style`	`Am_Style`	*Color of off pixels if opaque stipple*
`Am_IMAGE`	`Am_No_Image`	`Am_Image_Array`	*Stipple pattern*

The formulas in the `Am_WIDTH` and `Am_HEIGHT` slots reevaluate whenever a new image is installed in the `Am_IMAGE` slot. If a destructive modification is made to the image class currently installed in the slot, then `Note_Changed()` will have to be called to cause the formulas to reevaluate and to cause the object to be redrawn (for details about destructive modification of slot values and formula evaluation, see the ORE chapter).

`Am_Image_Array` is a regular C++ class, defined in `gdefs.h`. Here is a list of the member functions that are available for the `Am_Image_Array` class. The creator functions are used to make customized instances containing images described by data arrays or data stored in files. The other functions are for accessing, changing, and saving images to a file. Section 4.5.7.2 contains an example of how to use the `Am_Image_Array` class with the `Am_Bitmap` object. Section 4.6.3.7 discusses how to use the `Am_Image_Array` class with `Am_Style`.

```
Am_Image_Array (int percent);                  // Halftone pattern (see Section 4.6.2.2)

Am_Image_Array (unsigned int width,            // Solid rectangular image
                unsigned int height,
                int depth,
                Am_Style intial_color);
Am_Image_Array (const char* file_name);        // Image read from a file.

// The size of an image will be zero until drawn, & depends on the window in which the image is displayed.
void Get_Size (int& width, int& height);
```

4.5.7.1 Loading Am_Image_Arrays From a File

`Am_Image_Array` images can be loaded from several different image file formats depending on what platform you're using. Amulet decides which type of image to load based on the filename's suffix.

Full color GIF images are available on all three platforms. Files with names ending in 'gif' will be loaded as GIF images. Both 87a and 89a formats are supported, but only the first image in a file is loaded. Amulet reads the GIF colormap from the file, and provides the closest colors available on the display's current color map. Amulet does not install its own colormap, and Amulet does not support changing colormaps or image colors.

On Unix/ X machines, all GIF images are treated as multiple bit depth pixmaps (except on monochrome displays). You cannot specify the foreground and background pixel values for these images, and you cannot draw them transparent. You cannot specify them as stipple fill styles.

PC BMP bitmap images are available only on Windows NT/ '95 machines. On the PC, files with names ending in 'bmp' are loaded as Windows bitmap files.

XWindows XBM bitmap format is available only on Unix/ X machines. Under Unix, files with names that do not end in 'gif' are assumed to be XBM format bitmaps. These bitmaps are loaded as single bit pixmaps under X, and can be used as stipple fill styles. Specifying foreground and background colors explicitly, and creating transparent bitmaps are possible with XBM format images.

To make your code as machine-independant as possible, you should use GIF image format whenever feasible to prevent conditional compilation of different image filenames. For code examples of using `Am_Bitmap` with these file formats, see the `space` sample program in your Amulet directory.

4.5.7.2 Using Images with Am_Bitmap

To display an image whose description is stored in a file, you must first create an instance of `Am_Image_Array` initialized with the name of the file, and then install that image in the `Am_IMAGE` slot of your `Am_Bitmap` object.

4.6 Styles

The C++ class `Am_Style` is used to specify a set of graphical properties, such as color and line thickness. The `Am_LINE_STYLE` and `Am_FILL_STYLE` slots present in all graphical objects hold instances of `Am_Style`. The definition of the `Am_Style` class is in `gdefs.h`, and the predefined styles are listed in `opal.h`.

There are many predefined styles, like Am_Red, that provide the most common colors and thicknesses. You can also create your own custom styles by calling the constructor functions for Am_Style with your desired parameters. Whether you use a predefined style or create your own, you can set it directly into the appropriate slot of a graphical object. The style placed in the Am_LINE_STYLE slot of a graphical object controls the drawing of lines and outlines, and the style in the Am_FILL_STYLE slot controls the inside of the object.

Instances of Am_Style are immutable, and cannot be changed after they are created.

4.6.1 Predefined Styles

The most frequently used styles are predefined by Amulet. You can use any of the styles listed in this section directly in the Am_LINE_STYLE or Am_FILL_STYLE slot of an object.

The following color styles have line thickness zero (which really means 1, as explained in Section 4.6.3.2)

Am_Red	Am_Cyan	Am_Motif_Gray	Am_Motif_Light_Gray
Am_Green	Am_Orange	Am_Motif_Blue	Am_Motif_Light_Blue
Am_Blue	Am_Black	Am_Motif_Green	Am_Motif_Light_Green
Am_Yellow	Am_White	Am_Motif_Orange	Am_Motif_Light_Orange
Am_Purple		Am_Amulet_Purple	

The following styles are black.

Am_Thin_Line	Am_Line_1	Am_Line_4	Am_Dashed_Line
Am_Line_0	Am_Line_2	Am_Line_8	Am_Dotted_Line

The following styles are all black transparent or black and white opaque fills

Am_Gray_Stipple	Am_Opaque_Gray_Stipple
Am_Light_Gray_Stipple	Am_Diamond_Stipple
Am_Dark_Gray_Stipple	Am_Opaque_Diamond_Stipple

4.6.2 Creating Simple Line and Fill Styles

4.6.2.1 Thick Lines

To quickly create black line styles of a particular thickness, you can use the following special Am_Style creator function:

```
Am_Style::Thick_Line (unsigned short thickness);
```

For example, if you wanted to create a black line style 5 pixels thick, you could say "`black5 = Am_Style::Thick_Line (5)`". To specify the color or any other property simultaneously with the thickness, you have to use the full Am_Style creator functions discussed in Section 4.6.3.

4.6.2.2 Halftone Stipples

Stippled styles repeat a pattern of "on" and "off" pixels throughout a line style or fill style. A *halftone* is the most common type of stipple pattern, where the "on" and "off" bits are regularly spaced to create darker or lighter shades of a color. When mixing black and white pixels, for example, a 50% stipple of black and white bits will look gray. A 75% stipple will look darker, and a 25% stipple will look lighter. Some gray stipples are predefined in Amulet, and listed in Section 4.6.1. More complicated stipples, such as diamond patterns, are discussed in Section 4.6.3.7.

To create a simple halftone style with a regular stipple pattern, use this special `Am_Style` creator function:

```
Am_Style::Halftone_Stipple (int percent,
          Am_Fill_Solid_Flag fill_flag = Am_FILL_STIPPLED);
```

The *percent* parameter determines the shade of the halftone (0 is white and 100 is black). The *fill_flag* determines whether the pattern is transparent or opaque (see Section 4.6.3.6). In order to create a halftone that is one-third black and two-thirds white, you could say "`gray33 = Am_Style::Halftone_Stipple (33)`". There are only 17 different halftone shades available in Amulet, so several values of *percent* will map onto each built-in shade.

4.6.3 Customizing Line and Fill Style Properties

Any property of a style can be specified by creating an instance of `Am_Style`. The properties are provided as parameters to the `Am_Style` constructor functions. All the parameters have convenient defaults, so you only have to specify values for the parameters you are interested in. Since styles are used for both line styles and fill styles, some of the parameters only make sense for one kind of style or the other. The parameters that do not apply in a particular drawing situation are silently ignored.

```
Am_Style (float red, float green, float blue,          //color part
        short thickness = 0,
        Am_Line_Cap_Style_Flag cap_flag = Am_CAP_BUTT,
        Am_Join_Style_Flag join_flag = Am_JOIN_MITER,
        Am_Line_Solid_Flag line_flag = Am_LINE_SOLID,
        const char* dash_list = Am_DEFAULT_DASH_LIST,
        int dash_list_length = Am_DEFAULT_DASH_LIST_LENGTH,
        Am_Fill_Solid_Flag fill_flag = Am_FILL_SOLID,
        Am_Fill_Poly_Flag poly_flag = Am_FILL_POLY_EVEN_ODD,
        Am_Image_Array stipple = Am_No_Image)

Am_Style (const char* color_name,                      //color part
        short thickness = 0,
        Am_Line_Cap_Style_Flag cap_flag = Am_CAP_BUTT,
        Am_Join_Style_Flag join_flag = Am_JOIN_MITER,
        Am_Line_Solid_Flag line_flag = Am_LINE_SOLID,
        const char *dash_list = Am_DEFAULT_DASH_LIST,
        int dash_list_length = Am_DEFAULT_DASH_LIST_LENGTH,
        Am_Fill_Solid_Flag fill_flag = Am_FILL_SOLID,
        Am_Fill_Poly_Flag poly_flag = Am_FILL_POLY_EVEN_ODD,
        Am_Image_Array stipple = Am_No_Image)

static Am_Style Thick_Line (unsigned short thickness);
```

```
static Am_Style Halftone_Stipple (int percent,
        Am_Fill_Solid_Flag fill_flag = Am_FILL_STIPPLED);
```

The only required parameters for these style constructors are for the colors. Before you read the details below about what all the other parameters mean, be aware that most applications will just use the default values.

4.6.3.1 Color Parameter

The `Am_Style` constructor functions allow color to be specified in two ways: either with *red, green,* and *blue* values, or with a *color_name* such as "pink". The RGB values should be `floats` between `0.0f` and `1.0f`, where `1.0f` is full on. Gem maintains a table of color names and their corresponding red, green, and blue values. In X, all standard X color names on your server are supported. A small common subset was chosen on the Mac and PC to get color indices.

Amulet does not install its own colormap, to be friendlier to other applications on your system. Not all colors will be available on your platform. When a color is not available on your machine, Amulet finds the closest available color, and uses that instead. On Unix/ X, a warning is printed to your console saying that a color cell could not be allocated. Quitting applications that use many colors will help Amulet applications get the colors they need. On the PC, Windows automatically dithers solid colors together to get a closer match than possible with a single color. This sometimes produces undesirable results, but it is unavoidable.

4.6.3.2 Thickness Parameter

The *thickness* parameter holds the integer line thickness in pixels. Zero thickness lines are drawn with line thickness one. On some platforms, there may be a subtle difference between lines with thickness zero and lines with thickness one. Zero thickness lines might be drawn on some platforms with a more efficient device-dependent line drawing algorithm than is used for lines with thickness of one or greater. For this reason, a thickness zero line parallel to a thick line may not be as aesthetically pleasing as a line with thickness one.

For instances of `Am_Rectangle`, `Am_Roundtangle`, and `Am_Arc`, increasing the thickness of the line style will not increase the width or height of the object. The object will stay the same size, but the colored boundary of the object will extend *inwards* to occupy more of the object. Increasing the thickness of the line style of an `Am_Line` or `Am_Polygon` (objects which calculate their bounding box, instead of having it provided by the user) will increase the object's width and height. For these objects the thickness will extend *outward* on *both sides* of the line or polyline.

4.6.3.3 Cap_Flag Style Parameter

The *cap_flag* parameter determines how the end caps of line segments are drawn in X11. This parameter is ignored on the PC and Mac. Allowed values are elements of the enumerated type `Am_Line_Cap_Style_Flag`:

cap_flag	Result
`Am_CAP_BUTT`	Square at the endpoint (perpendicular to the slope of the line) with no projection beyond.
`Am_CAP_NOT_LAST`	Equivalent to `Am_CAP_BUTT`, except that for thickness 0 or 1 the final endpoint is not drawn.
`Am_CAP_ROUND`	A circular arc with the diameter equal to the thickness centered on the endpoint.
`Am_CAP_PROJECTING`	Square at the end, but the path continues beyond the endpoint for a distance equal to half of the thickness.

4.6.3.4 Join_Flag Style Parameter

The *join_flag* parameter determines how corners (where multiple lines come together) are drawn for thick lines as part of rectangle and polygon objects in X11. This parameter is ignored on the PC. This does not affect individual lines (instances of `Am_Line`) that are part of a group, even if they happen to have the same endpoints. Allowed values are elements of the enumerated type `Am_Join_Style_Flag`:

join_flag	Result
`Am_JOIN_MITER`	The outer edges of the two lines extend to meet at an angle.
`Am_JOIN_ROUND`	A circular arc with a diameter equal to the *thickness* is drawn centered on the join point.
`Am_JOIN_BEVEL`	Endpoints of lines are drawn with `Am_CAP_BUTT` style, with the triangular notch filled.

4.6.3.5 Dash Style Parameters

The *line_flag* parameter determines whether the line is solid or dashed, and how the spaces between the dashes should be drawn. Valid values are elements of the enumerated type `Am_Line_Solid_Flag`:

line_flag	Result
`Am_LINE_SOLID`	No dashes
`Am_LINE_ON_OFF_DASH`	Only the "on" dashes are drawn, and nothing is drawn in the "off" dashes.

The *dash_list* and *dash_list_length* parameters describe the pattern for dashed lines. The *dash_list* should be a `const char*` array that holds numbers corresponding to the pixel length of the "on" and "off" pixels. The default `Am_DEFAULT_DASH_LIST` value is `{4 4}`. A *dash_list* of `{1 1 1 1 3 1}` is a typical dot-dot-dash line. A list with an odd number of elements is equivalent to the list being appended to itself. Thus, the *dash_list* `{3 2 1}` is equivalent to `{3 2 1 3 2 1}`.

The following code defines a dash pattern with each "on" and "off" dash 15 pixels long. To see the result of this code, store the `thick_dash` style in the `Am_LINE_STYLE` slot of a graphical object.

```
static char thick_dash_list[2] = {15, 15};
Am_Style thick_dash ("black", 8, Am_CAP_BUTT, Am_JOIN_MITER,
                     Am_LINE_ON_OFF_DASH, thick_dash_list);
```

Dashed and dotted lines are not supported on the Macintosh. In a future version of Amulet, we will either support the current implementation of dashed and dotted lines on all platforms, or more likely, we'll provide a few predefined dashed and dotted line styles. The dotted and dashed line styles are guaranteed to look different from each other and from normal lines, but we will not provide as much support for complex dashed lines. To maintain forward compatability, you should only use the predefined dotted and dashed line styles `Am_Dashed_Line` and `Am_Dotted_Line`.

4.6.3.6 Fill Style Parameters

The *fill_flag* determines the way "off" pixels in the stippled pattern (see Section 4.6.3.7) will be drawn. The "on" pixels are always drawn with the color of the style. Allowed values are elements of the enumerated type `Am_Fill_Solid_Flag`:

fill_flag	Result
`Am_FILL_SOLID`	Draw "off" pixels same as "on" pixels.
`Am_FILL_TILED`	Not implemented yet
`Am_FILL_STIPPLED`	Only the "on" pixels are drawn, and nothing is drawn for the "off" pixels (transparent stipple).
`Am_FILL_OPAQUE_STIPPLED`	Draw the "off" pixels in white.

The value of the *poly_flag* parameter should be an element of the enumerated type `Am_Fill_Poly_Flag`, either `Am_FILL_POLY_EVEN_ODD` or `Am_FILL_POLY_WINDING`. This parameter controls the filling for self-intersecting polygons, like the five-pointed star example in Section 4.5.5.2.

4.6.3.7 Stipple Parameters

A stippled style consists of a small pattern of "on" and "off" pixels that is repeated throughout the border or filling of an object. The simplest stipple pattern is the halftone, discussed in Section 4.6.2.2. You should only need to specify the *stipple* parameter in the full `Am_Style` creator functions when you are specifying some other property (like color) along with a non-solid stipple, or you are specifying an unconventional image for your stipple pattern.

The value of the *stipple* parameter should be an instance of `Am_Image_Array`. An image array holds the pattern of bits, which can either be a standard halftone pattern or something more exotic. The creator functions and other member functions for `Am_Image_Array` are discussed in Section 4.5.7.1.

On Unix/X, only XBM images can be used as stipples. GIF stipples and transparent GIFs are not currently supported on Unix.

Here is an example of a colored style with a 50% halftone stipple, created using the halftone initializer for `Am_Image_Array`:

```
Am_Style red_stipple ("red", 8, Am_CAP_BUTT, Am_JOIN_MITER, Am_LINE_SOLID,
                      Am_DEFAULT_DASH_LIST, Am_DEFAULT_DASH_LIST_LENGTH,
                      Am_FILL_STIPPLED, Am_FILL_POLY_EVEN_ODD,
                      (Am_Image_Array (50)) );
```

Here is an example of a stipple read from a file. The "stripes" file contains a description of a bitmap image. On Unix, the "stripes" file must be in X11 bitmap format (i.e., generated with the Unix `bitmap` utility). On the PC, the "stripes" file must be in either `BMP` or `GIF` format. This means that for portable code, you will need to use the `#ifdef` macro to load different files depending on your platform.

```
Am_Style striped_style ("black", 8, Am_CAP_BUTT, Am_JOIN_MITER,
                        Am_LINE_SOLID, Am_DEFAULT_DASH_LIST,
                        Am_DEFAULT_DASH_LIST_LENGTH, Am_FILL_STIPPLED,
                        Am_FILL_POLY_EVEN_ODD,
                        (Am_Image_Array ("stripes")) );
```

4.6.4 Getting Style Properties

You can query certain properties of an already-created style. This is useful if you want to create a style with the same color as another style, but with a different line thickness, for example.

Here are the `Am_Style` methods available for getting the values of a style's properties:

```
void Get_Values (float& red, float& green, float& blue) const;
void Get_Values (short& thickness,
        Am_Line_Cap_Style_Flag& cap, Am_Join_Style_Flag& join,
        Am_Line_Solid_Flag& line_flag,const char*& dash_l,
        int& dash_l_length, Am_Fill_Solid_Flag& fill_flag,
        Am_Fill_Poly_Flag& poly, Am_Image_Array& stipple) const;

void Get_Values (float& r, float& g, float& b, short& thickness,
        Am_Line_Cap_Style_Flag& cap, Am_Join_Style_Flag& join,
        Am_Line_Solid_Flag& line_flag, const char*& dash_l,
        int& dash_l_length,Am_Fill_Solid_Flag& fill_flag,
        Am_Fill_Poly_Flag& poly,Am_Image_Array& stipple) const;
```

```
Am_Fill_Solid_Flag Get_Fill_Flag() const;
Am_Image_Array Get_Stipple() const;
Am_Fill_Poly_Flag Get_Fill_Poly_Flag () const;

//Get the properties needed to calculate the line width
void Get_Line_Thickness_Values (short& thickness,
        Am_Line_Cap_Style_Flag& cap) const;

const char* Get_Color_Name () const; //returns a pointer to the string, don't dealloc
```

4.7 Groups

Groups hold a collection of graphical objects (possibly including other groups). The objects in a group are called its *parts*, and the group is the *owner* of each of its parts. The concept of part/owner relationships was introduced in the ORE chapter, but groups treat their parts specially, by drawing them in windows. In Opal, the part-owner hierarchy corresponds to a graphical hierarchy, when graphical objects are involved.

Here are the slots of Am_Group:

Slot	Default Value	Type	
Am_VISIBLE	true	bool	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	10	int	
Am_HEIGHT	10	int	
Am_GRAPHICAL_PARTS	empty Am_Value_List	Am_Value_List	*Read only*
Am_X_OFFSET	0	int	
Am_Y_OFFSET	0	int	
Am_H_SPACING	0	int	
Am_V_SPACING	0	int	
Am_H_ALIGN	Am_CENTER_ALIGN	{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}	
Am_V_ALIGN	Am_CENTER_ALIGN	{Am_TOP_ALIGN, Am_BOTTOM_ALIGN, Am_CENTER_ALIGN}	
Am_FIXED_WIDTH	Am_NOT_FIXED_SIZE	int	
Am_FIXED_HEIGHT	Am_NOT_FIXED_SIZE	int	
Am_INDENT	0	int	
Am_MAX_RANK	false	int, bool	
Am_MAX_SIZE	false	int, bool	

Groups define their own coordinate system, meaning that the left and top of their parts is offset from the origin of the group. Changing the position of the group *translates* the position of all its parts. By using a special group called Am_Resize_Parts_Group, you can also have the group *scale* the size of its parts when the group's size changes.

Groups also *clip* their parts to the bounding box of the group. Parts of objects that extend outside the left, top, width, or height of the group are not drawn. The default width and height of a group is 10x10, so you must be careful to set the `Am_WIDTH` and `Am_HEIGHT` slot of your instances of `Am_Group`. The predefined constraints `Am_Width_Of_Parts` and `Am_Height_Of_Parts` may be used to compute the size of a group based on its parts

4.7.1 Adding and Removing Graphical Objects

The read only `Am_GRAPHICAL_PARTS` slot of a group contains an Am_Value_List of objects. It can be used to iterate over the graphical parts of an object. Parts of a group should be manipulated with the following member functions defined on `Am_Object`:

```
Am_Object Add_Part (Am_Object new_part, bool inherit = true);
Am_Object Add_Part (Am_Slot_Key key, Am_Object new_part);

void Remove_Part (Am_Slot_Key key);
void Remove_Part (Am_Object part);
```

Graphical parts added to an object with `Add_Part` will automatically be added to the `Am_GRAPHICAL_PARTS` list of the object.

Parts of an object may or may not be inherited (instantiated) in an instance of that object. All parts added with a slot key (using the second form of `Add_Part`, above) to an owner are instantiated in instances of the owner. Parts added without a slot key (using the first form of `Add_Part`, above) are only instantiated in instances of their owner if the second parameter, inherit, is true (the default is true).

Parts that are added with a slot key can be accessed with `Get_Part()` or `Get()`. It is often convenient to provide slot keys for parts so that functions and formulas can easily access these objects in their groups. All graphical parts can be accessed through the `Am_GRAPHICAL_PARTS` list.

You can add any kind of object (graphical, non-graphical, windows, screens) as a part of any other object. However, things only "work right" if graphical objects are added to groups or windows, which are added to windows or the screen.

4.7.2 Layout

Most groups do not use a layout procedure. In these groups, each part has its own left and top, which places it at some user-defined position relative to the left and top of the group.

It is sometimes convenient for the group itself to decide where its graphical parts should be located. If the parts should all be in a row or column, it's often easier and more extensible to tell the group that all of its parts should be arranged in a specific way, than to try to keep track of the locations of all of the objects in the group individually. To support this, a formula in the Am_LAYOUT slot of an Am_Group object can lay out all of the parts of the group. This formula operates by directly setting the Am_LEFT and Am_TOP of the parts.

4.7.2.1 Vertical and Horizontal Layout

Amulet provides two built-in layout procedures:

```
Am_Formula  Am_Vertical_Layout
Am_Formula  Am_Horizontal_Layout
```

These layout procedures arrange the parts of a group according to the values in the slots listed below. To arrange the parts of a group in a vertical list (like a menu), set the Am_LAYOUT slot to Am_Vertical_Layout. You may then want to set other slots of the group, like Am_V_SPACING, to control things like the spacing between parts or the number of columns.

These procedures set values in the Am_LEFT and Am_TOP slots of the graphical parts of the group, overriding whatever values (or formulas) were there before.

The slots that control layout when using the standard vertical or horizontal layout procedures are:

- Am_X_OFFSET — The horizontal space to leave between the origin of the group and the first part that is placed, measured in number of pixels (default is 0)
- Am_Y_OFFSET — Same as Am_X_OFFSET, only vertical (default is 0)
- Am_H_SPACING — The horizontal space to leave between parts, measured in pixels (default is 0)
- Am_V_SPACING — Same as Am_H_SPACING, only vertical (default is 0)
- Am_H_ALIGN — Justification for parts within a column: when a narrow part appears in a column with other wider parts, this parameter determines whether the narrow part is positioned at the left, center, or right of the column (default is Am_CENTER_ALIGN)
- Am_V_ALIGN — Same as Am_H_ALIGN, only vertical (default is Am_CENTER_ ALIGN)
- Am_FIXED_WIDTH — The width of each column, probably based on the width of the widest part. When Am_NOT_FIXED_SIZE is used, the columns are not necessarily all the same width; instead, the width of each column is determined by the widest part in that column. (default is Am_NOT_FIXED_SIZE)
- Am_FIXED_HEIGHT — Same as Am_FIXED_WIDTH, only vertical (default is Am_NOT_FIXED_SIZE)
- Am_INDENT — How much to indent the second row or column (depending on horizontal or vertical orientation), measured in number of pixels (default is 0)
- Am_MAX_RANK — The maximum number of parts allowed in a row or column, depending on horizontal or vertical orientation (default is false)

- Am_MAX_SIZE The maximum number of pixels allowed for a row or column, depending on horizontal or vertical orientation (default is false)

The following will create a group with a column containing a rectangle and a circle:

```
Am_Object my_group = Am_Group.Create ("my_group")
    .Set (Am_LEFT, 10)
    .Set (Am_TOP, 10)
    .Set (Am_LAYOUT, Am_Vertical_Layout)
    .Set (Am_V_SPACING, 5)
    .Add_Part(Am_Rectangle.Create())
    .Add_Part(Am_Circle.Create());
```

4.7.2.2 Custom Layout Procedures

You can provide a customized layout procedure for arranging the parts of a group. The procedure should be defined as a constraint, using Am_Define_Formula or a related function, and the constraint should be installed in the Am_LAYOUT slot of the group. The parts of the group should be arranged as a side effect of evaluating the formula (the return value is ignored). To do this, GV the list in the Am_GRAPHICAL_PARTS slot (which is in Z-order) and iterate through it, setting each part's Am_LEFT and Am_TOP slots appropriately.

4.7.3 Am_Resize_Parts_Group

This group operates like a regular Am_Group except that if the width or height of the Am_Resize_Parts_Group is changed, then the width and height of all the components is scaled proportionately. The width and height of the Am_Resize_Parts_Group should *not* be a formula depending on the parts, and the parts should *not* have formulas in their width and height slots. Instead, the width and heights will usually be integers, with the original size of the group set to be the correct size based on the parts. Be sure to adjust the width and height of a Am_Resize_Parts_Group when new parts are added or removed. Since you cannot use formulas that depend on the parts' sizes, you must explicitly make sure the group is always big enough to hold its parts.

All of the parts of a Am_Resize_Parts_Group are expected to be able to resize themselves when their width and heights are set. It is fine to use a Am_Resize_Parts_Group in another one, but it would usually be an error to include a regular Am_Group inside a Am_Resize_Parts_Group.

The Am_Resize_Parts_Group is created automatically by the Am_Graphics_Group_Command when the user selects a number of objects and executes a "Group" command (see Section 6.4 in the Widgets chapter).

Note: the Am_Resize_Parts_Group currently scales the parts using integers, so if the size gets very small, the parts will not retain their original proportions, and if the size of the group goes to zero, the objects will stay small forever.

4.8 Maps

The Am_Map object is a special kind of group that generates multiple graphical parts from a single prototype object. Maps should be used when all the parts of a group are similar enough that they can be generated from one prototype object (for example, they are all rectangles, or all the same kind of group.). This part-generating feature of maps is often used in conjunction with the layout feature of groups, in a situation such as arranging the selectable text items in a menu. For details on laying out the components of groups and maps, see Section 4.7.2.

Slot	Default Value	Type
Am_VISIBLE	true	bool
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	Am_Width_Of_Parts	int
Am_HEIGHT	Am_Height_Of_Parts	int
Am_GRAPHICAL_PARTS	<formula>	Am_Value_List
Am_ITEMS	0	int, Am_Value_List
Am_ITEM_PROTOTYPE	Am_No_Object	Am_Object
Am_LAYOUT	NULL	<formula>
Am_X_OFFSET	0	int
Am_Y_OFFSET	0	int
Am_H_SPACING	0	int
Am_V_SPACING	0	int
Am_H_ALIGN	Am_CENTER_ALIGN	{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}
Am_V_ALIGN	Am_CENTER_ALIGN	{Am_TOP_ALIGN, Am_BOTTOM_ALIGN, Am_CENTER_ALIGN}
Am_FIXED_WIDTH	Am_NOT_FIXED_SIZE	int
Am_FIXED_HEIGHT	Am_NOT_FIXED_SIZE	int
Am_INDENT	0	int
Am_MAX_RANK	false	int, bool
Am_MAX_SIZE	false	int, bool

You must set two slots in the map to control the parts that are generated:

- Am_ITEMS The value should be either a number, specifying how many parts should be generated, or an instance of Am_Value_List, containing elements corresponding to each part to be generated.
- Am_ITEM_PROTOTYPE A graphical group, to serve as the prototype for each part.

There are two slots automatically installed in each of the generated parts, that are useful for distinguishing the parts from each other. These slots can be referenced by formulas in the item prototype to make each part different.

- Am_RANK The position of this part in the list, from 0
- Am_ITEM The element of the map's Am_ITEMS list that corresponds to this part

The Am_RANK of each created part is set with the count of this part. The Am_RANK of the first part created is set to 0, the second part's Am_RANK is set to 1, and so on. If the Am_ITEMS slot of the map contains an Am_Value_List, then the Am_ITEM (note: singular) slot of each created part is set with the corresponding element of the list.

The following code defines a map whose Am_ITEMS slot is a number. The map generates 4 rectangles, whose fill styles are determined by the formula map_fill_from_rank. The formula computes a halftone fill from the value stored in the Am_RANK slot of the part, which was installed by the map as the part was created. This uses a horizontal layout formula so the rectangles will be in a row.

```
// Formulas are defined in the global scope, outside of main()
Am_Define_Style_Formula (map_fill_from_rank) {
  int rank = self.GV (Am_RANK);
  return Am_Style::Halftone_Stipple (20 * rank);
}
...

  // This code is inside main()
  Am_Object my_map = Am_Map.Create ("my_map")
    .Set (Am_LEFT, 10)
    .Set (Am_TOP, 10)
    .Set (Am_LAYOUT, Am_Horizontal_Layout)
    .Set (Am_H_SPACING, 5)
    .Set (Am_ITEMS, 4)
    .Set (Am_ITEM_PROTOTYPE, Am_Rectangle.Create ("map item")
          .Set (Am_FILL_STYLE, map_fill_from_rank)
          .Set (Am_WIDTH, 20)
          .Set (Am_HEIGHT, 20));
```

The next example defines a map whose Am_ITEMS slot contains a list of strings. The map generates 4 text objects, whose text strings are determined by the object's Am_ITEM slot.

```
// This code is inside main()
Am_Object my_map = Am_Map.Create ("my_map")
  .Set (Am_LEFT, 10)
  .Set (Am_TOP, 10)
  .Set (Am_LAYOUT, Am_Vertical_Layout)
  .Set (Am_V_SPACING, 5)
  .Set (Am_ITEMS, Am_Value_List ()
        .Add ("This is the first item in the map.")
        .Add ("I'm number two")
        .Add ("Three")
        .Add ("The last item in the list."))
  .Set (Am_ITEM_PROTOTYPE, Am_Text.Create ("map text item")
        .Set (Am_ITEM, "") // initialize the slot so the formula won't crash
        .Set (Am_TEXT, Am_Same_As(Am_ITEM))
       );
```

To add another item to the map in the second example, you could install a new list in the Am_ITEMS slot containing all the old items plus the new one:

```
my_map.Make_Unique (Am_ITEMS); //in case slot shared with another object
Am_Value_List map_items = (Am_Value_List) my_map.Get (Am_ITEMS);
map_items.Add ("A new item!");
my_map.Set (Am_ITEMS, map_items);
```

A more efficient way to add an item to the list is to destructively modify the list that is already installed (note the use of the *false* parameter in the Add method for Am_Value_List):

```
Am_Value_List map_items = (Am_Value_List) my_map.Get (Am_ITEMS);
map_items.Add ("A new item!", false); //false means destructively modify, don't copy.
my_map.Note_Changed (Am_ITEMS);
```

The list in the Am_ITEMS slot can also be calculated with a formula, and the items in the map will change whenever the formula is reevaluated.

4.9 Methods on all Graphical Objects

4.9.1 Reordering Objects

As you add objects to a group or window, each new object by default is on top of the previous one. This is called the "Z" or "stacking" or "covering" order.

The following functions are useful for changing the Z order of an object among its siblings. For example, Am_To_Top(obj) will bring an object to the front of all of the other objects in the same group or window. To promote an object just above a certain target object, use Am_Move_Object (obj, target_obj, true). These functions work for windows as well as for regular graphical objects.

```
void Am_To_Top (Am_Object object);
void Am_To_Bottom (Am_Object object);
void Am_Move_Object (Am_Object object, Am_Object ref_object,
                     bool above = true); //false means below ref_object
```

4.9.2 Finding Objects from their Location

The following functions are useful for determining whether an object is under a given (x,y) coordinate:

```
bool Am_Point_In_All_Owners(Am_Object in_obj, int x, int y,
                            Am_Object ref_obj);

Am_Object Am_Point_In_Obj (Am_Object in_obj, int x, int y,
                           Am_Object ref_obj);

Am_Object Am_Point_In_Part (Am_Object in_obj, int x, int y,
                            Am_Object ref_obj,
                            bool want_self = false,
                            bool want_groups = true);

Am_Object Am_Point_In_Leaf (Am_Object in_obj, int x, int y,
```

```
                                    Am_Object ref_obj,
                                    bool want_self = true,
                                    bool want_groups = true);
```

`Am_Point_In_All_Owners` checks whether the point is inside all the owners of object, up to the window. Also validates that all of the owners are visible. If not, returns false. Use this to make sure that the user is not pressing outside of an owner since the other operations do *not* check this.

`Am_Point_In_Obj` checks whether the point is inside the object. It ignores covering (i.e., it just checks whether point is inside the object, even if the object is behind another object). If the point is inside, the object is returned; otherwise the function returns `Am_No_Object`. The coordinate system of x and y is defined with respect to `ref_obj`, that is, the origin of x and y is the left and top of `ref_obj`.

`Am_Point_In_Part()` finds the front-most (least covered) immediate part of `in_obj` at the specified location. If there is no part at that point, it returns `Am_No_Object`. The coordinate system of *x* and *y* is defined with respect to *ref_obj*. If there is no part at that point, then if `want_self` then if inside `in_obj`, returns `in_obj`. If *not* `want_self` or *not* inside `in_obj`, returns `Am_No_Object`. The coordinate system of x and y is defined with respect to *ref_obj*. If `want_groups` is true, then returns the part even if it is a group. If `want_groups` is false, then will not return a group (so if x,y is not over a "primitive" object, returns `Am_No_Object`).

`Am_Point_In_Leaf()` is similar to `Am_Point_In_Part()`, except that the search continues to the deepest part in the group hierarchy (i.e., it finds the leaf-most object at the specified location). If (*x*,*y*) is inside the bounding box of `in_obj` but not over a leaf, it returns *in_obj*. The coordinate system of *x* and *y* is defined with respect to *ref_obj*. Sometimes you will want a group to be treated as a leaf in this search even though it isn't really a leaf. In this case, you should set the `Am_PRETEND_TO_BE_LEAF` slot to true for each group that should be treated like a leaf. The search will not look through the parts of such a group, but will return the group itself. The slots `want_self` and `want_groups` work the same as for `Am_Point_In_Part`.

`Am_Point_In_Part()` and `Am_Point_In_Leaf()` use the function `Am_Point_In_Obj()` on the parts.

4.9.3 Beeping

```
void Am_Beep (Am_Object window = Am_No_Object);
```

This function causes the computer to emit a "beep" sound. Passing a specific window is useful in Unix, when several different screens might be displaying windows, and you only want a particular screen displaying a particular window to beep.

4.9.4 Filenames

```
char *Am_Merge_Pathname (char *name);
```

`Am_Merge_Pathname()` takes a filename as a parameter, and returns the full Amulet directory pathname prepended to that argument. For example, "`Am_Merge_Pathname ("lib/images/ent.bmp")`" will return the full pathname to the PC compatible Enterprise bitmap included with the Amulet source files.

On the Macintosh, Am_Merge_Pathname automatically converts the Unix-standard path separation character "/" into the Mac-specific path separator ":". In Windows NT/'95, this conversion is done automatically by the OS. On all systems, you should specify pathnames with slashes ("/") as the path separator to avoid machine dependancy.

4.9.5 Translate Coordinates

```
bool Am_Translate_Coordinates (Am_Object src_obj, int src_x, int src_y,
                               Am_Object dest_obj, int& dest_x, int& dest_y,
                               Am_Constraint_Context& cc = *Am_Empty_Constraint_Context);
```

`Am_Translate_Coordinates()` converts a point in one object's coordinate system to that of another object. It works for windows and all graphical objects. If the objects are not comparable (windows on separate screens, or windows not attached to any screen) then the function will return `false`. Otherwise, it will return `true` and *dest_x* and *dest_y* will contain the converted coordinates.

The destination coordinates are for the inside of *dest_obj*. This means that if *obj* was at *src_x*, *src_y* with respect to the left and top of *src_obj*, and you remove it from *src_obj* and add it to *dest_obj* at *dest_x*, *dest_y* then it will be at the same physical screen position. You can provide an `Am_Constraint_Context` parameter inside a formula to make the formula dependent on the relative positions of the objects.

Since each group and window defines its own coordinate system, you must use `Am_Translate_Coordinates` whenever you define a formula that depends on the left or top of an object that might be in a different group or window.

4.10 Windows

Objects are added to windows in the same way they're added to groups, with `Add_Part()`. All graphical objects added to a window will be displayed in that window. When a window is added as a part to another window, it becomes a *subwindow*. Subwindows do not have any window manager decoration (title bars).

Slot	Default Value	Type	
Am_VISIBLE	true	bool	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	100	int	
Am_HEIGHT	100	int	
Am_GRAPHICAL_PARTS	empty Am_Value_List	Am_Value_List	*read only*
Am_FILL_STYLE	Am_White	Am_Style	
Am_MAX_WIDTH	0	int	
Am_MAX_HEIGHT	0	int	
Am_MIN_WIDTH	1	int	
Am_MIN_HEIGHT	1	int	
Am_TITLE	"Amulet"	char*	
Am_ICON_TITLE	"Amulet"	char*	
Am_ICONIFIED	false	bool	
Am_USE_MIN_WIDTH	false	bool	
Am_USE_MIN_HEIGHT	false	bool	
Am_USE_MAX_WIDTH	false	bool	
Am_USE_MIN_HEIGHT	false	bool	
Am_QUERY_POSITION	false	bool	
Am_QUERY_SIZE	false	bool	
Am_IS_COLOR	formula	bool	*read only*
Am_OMIT_TITLE_BAR	false	bool	
Am_CLIP_CHILDREN	false	bool	
Am_DESTROY_WINDOW_ METHOD	Am_Default_Window_ Destroy_Method	Am_Object_ Method	
Am_DOUBLE_BUFFER	true	bool	
Am_SAVE_UNDER	false	false	

4.10.1 Slots of Am_Window

The initial values of Am_LEFT, Am_TOP, Am_WIDTH, and Am_HEIGHT determine the size and position of the window when it appears on the screen. These slots can be set later to change the window's size and position. If the user changes the size or position of a window using the window manager (e.g., using the mouse), this will be reflected in the values for these slots.

Note that under Unix/X, it's not always possible to know exactly where a window is on the screen. Some window managers specify screen position as the location of the titlebar, some specify it as the location of the client region, and some allow the user to choose the coordinate reference system. It's impossible for Amulet to enumerate all the possible things that a window manager might do, and take them into account. In this case, our goal is to have code that never breaks and that maintains internal consistency.

The Am_FILL_STYLE determines the background color of the window. All parameters of Am_Style that affect fillings, including stipples, affect the fill style of windows. Using the fill style of a window is more efficient than putting a window-sized rectangle behind all the other objects in the window.

When values are installed in the Am_MAX_WIDTH, Am_MAX_HEIGHT, Am_MIN_WIDTH, or Am_MIN_HEIGHT slots, and the corresponding Am_USE_MAX_WIDTH, Am_USE_MAX_HEIGHT, Am_USE_MIN_WIDTH, or Am_USE_MIN_HEIGHT slot is set to true, then the window manager will make sure the user is not allowed to change the window's size to be outside of those ranges. You can still set the Am_WIDTH and Am_HEIGHT to be any value, but the window manager will eventually clip them back into the allowed range.

When Am_QUERY_POSITION or Am_QUERY_SIZE are set to true, then the user will have the opportunity to place the window on the screen when the window is first added to the screen, clicking the left mouse button to position the left and top of the window, and dragging the mouse to the desired width and height.

The border widths applied to the window by the window manager are stored in the Am_LEFT_BORDER_WIDTH, Am_TOP_BORDER_WIDTH, Am_RIGHT_BORDER_WIDTH, and Am_BOTTOM_BORDER_WIDTH. These slots are read only, set by Amulet when the window becomes visible on the screen.

The Am_OMIT_TITLE_BAR slot tells whether the Amulet window should have a title bar. If the slot has value false (the default), and the window manager permits it, then the window will have a title bar; otherwise the window will not have a title bar.

In the rare case when you want to have graphics drawn on a parent window appear over the enclosed (child) windows, you can set the Am_CLIP_CHILDREN slot of the parent to be true. Then any objects that belong to that window will appear on top of the window's subwindows (rather than being hidden by the subwindows).

When the Am_DOUBLE_BUFFER slot is set to true (the default), Opal updates the window by first drawing everything offscreen, and then copying the region over the old contents of the window. This is slightly slower than updating without double buffering and it uses a lot of memory, because you need to copy the contents of the offscreen buffer back to the screen. However, using double buffering is much more visually pleasing, and flicker is reduced tremendously, because you don't see each object being drawn to the screen in succession; it all appears at once.

4.10.2 Destroying windows

Users might destroy Amulet windows using the window manager's Kill-Window command or close box. This action is converted into a destroy message to the window. By default this deletes the window, but you can override the default with other behaviors by providing a new destroy method. The Am_DESTROY_WINDOW_METHOD slot of Am_WINDOW holds an Am_Object_Method that is called when the window is destroyed. The default method is:

- Am_Default_Window_Destroy_Method, which destroys the window and exits the main event loop (causes the application to quit) if no windows are left.

Other predefined methods may be useful for various applications (defined in opal.h):

- Am_Window_Hide_Method makes the window be invisible and does not destroy it, which is useful for dialog boxes.
- Am_Window_Destroy_And_Exit_Method always destroys the window and quits the application. This might be useful for an application's main window.

To actually destroy a window object, just use the regular object destroy method: win.Destroy(). This does not cause the Am_DESTROY_WINDOW_METHOD to be called. That is only called when the user destroys the window using the window manager's commands.

4.11 Am_Screen

Windows are not visible until they are added to the screen. The Am_Screen object can be thought of as a root window to which all top-level windows are added. In the "hello world" example of Section 4.3.2, the top-level window is added to Am_Screen with a call to Add_Part().

Am_Screen can be used in calls to Am_Translate_Coordinates() to convert from window coordinates to screen coordinates and back again.

4.12 Predefined formula constraints

Opal provides a number of constraints that can be put into slots of objects that might be useful. Some of these constraints were described in previous sections.

Am_Fill_To_Bottom - Put in an object's Am_HEIGHT slot, causes the object to size itself so it's tall enough to fill to the bottom of its owner. Am_Fill_To_Bottom leaves a border below the object, with a size equal to the object's Am_Y_OFFSET slot.

`Am_Fill_To_Right` - Analogous to `Am_Fill_To_Bottom`, used in the `Am_WIDTH` slot of an object. The `Am_X_OFFSET` slot of the object is used to measure the border to the right of the object.

`Am_Width_Of_Parts` - Useful for computing the width of a group: returns the distance between the group's left and the right of its rightmost part. You might put this into a group's `Am_WIDTH` slot.

`Am_Height_Of_Parts` - Analogous to `Am_Width_Of_Parts`, but for the `Am_HEIGHT` of a group.

`Am_Right_Is_Right_Of_Owner` - Useful for keeping a part at the right of its owner. Put this formula in the `Am_LEFT` slot of the part.

`Am_Bottom_Is_Bottom_Of_Owner` - Useful for keeping a part at the bottom of its owner. Put this formula in the `Am_TOP` slot of the part.

`Am_Center_X_Is_Center_Of_Owner` - Useful for centering a part horizontally within its owner. Put this formula in the `Am_LEFT` slot of the part.

`Am_Center_Y_Is_Center_Of_Owner` - Useful for centering a part vertically within its owner. Put this formula in the `Am_TOP` slot of the part.

`Am_Center_X_Is_Center_Of` - Useful for horizontally centering `obj1` inside `obj2`. Put this formula in the `Am_LEFT` slot of `obj1`, and put `obj2` in the `Am_CENTER_X_OBJ` slot of `obj1`.

`Am_Center_Y_Is_Center_Of` - Useful for vertically centering `obj1` inside `obj2`. Put this formula in the `Am_TOP` slot of `obj1`, and put `obj2` in the `Am_CENTER_Y_OBJ` slot of `obj1`.

`Am_Horizontal_Layout` - Constraint which lays out the parts of a group horizontally in one or more rows. Put this into the `Am_LAYOUT` slot of a group.

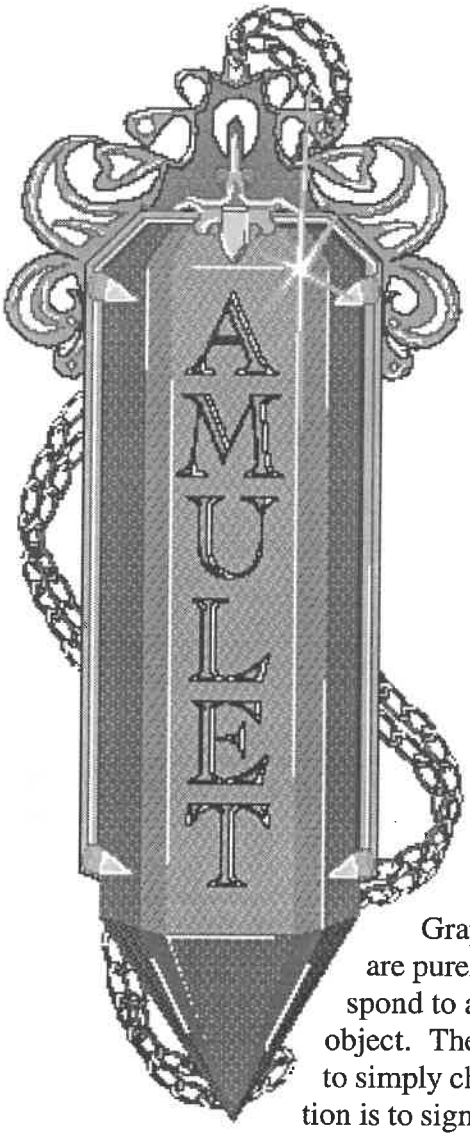
`Am_Vertical_Layout` - Constraint which lays out the parts of a group vertically in one or more columns. Put this into the `Am_LAYOUT` slot of a group.

`Am_Same_As (Am_Slot_Key key)` - This slot gets its value from the specified slot (`key`) in the same object. Equivalent to `{ return self.GV(key); }`

`Am_From_Owner (Am_Slot_Key key)` - This slot gets its value from the specified slot (`key`) in the object's owner. Equivalent to `{ return self.GV_Owner().GV(key); }`

`Am_From_Part (Am_Slot_Key part, Am_Slot_Key key)` - This slot gets its value from the specified slot (`key`) in the specified part (`part`) of this object. Equivalent to `{ return self.GV_Part(part).GV(key); }`

`Am_From_Sibling (Am_Slot_Key sibling, Am_Slot_Key key)` - This slot gets its value from the specified slot (`key`) in the specified sibling (`sibling`) of this object. Equivalent to `{ return self.GV_Sibling(sibling).GV(key); }`



5. Interactors and Command Objects for Handling Input

Graphical objects in Amulet do not respond to input events; they are purely output. When the programmer wants to make an object respond to a user action, an *Interactor* object is attached to the graphical object. The built-in types of Interactors usually enable the programmer to simply choose the correct type and fill in a few parameters. The intention is to significantly reduce the amount of coding necessary to define behaviors.

When an Interactor or a *widget* (see the Widgets chapter) finishes its operation, it allocates a *Command object* and then invokes the 'do' method of that Command object. Thus, the Command objects take the place of *call-back procedures* in other systems. The reason for having Command objects is that in addition to the 'do' method, a Command object also has methods to support undo, help, and selective enabling of operations. As with Interactors, Amulet supplies a library of Command objects so that often programmers can use a Command object from the library without writing any code.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

5.1 Include Files

The primary include files that control the Interactors and Command objects are `inter.h` for the main, top-level objects and procedures, `idefs.h` for the definitions specific to input events, and `inter_advanced.h` for what you might need if you are going to create your own custom Interactors. All the slots of the Interactor and Command objects are defined in `standard_slots.h`. Some of the functions and types needed to customize the text editing Interactor are defined in `text_fns.h`. For more information on the Amulet header files and how to use them, see Section 1.6 in the Overview chapter.

5.2 Overview of Interactors and Commands

The graphical objects created with Opal do not respond to input devices: they are just static graphics. In order to handle input from the user, you create an "Interactor" object and attach it to the graphics. The Interactor objects have built-in behaviors that correspond to the normal operations performed in direct manipulation user interfaces, so usually coding interactive interfaces is quick and easy using interactors. However, like programming with constraints, programming with Interactors requires a different "mind set" and the programming style is probably different than what most programmers are used to.

All of the Interactors are highly parameterized so that you can control many aspects of the behavior simply by setting slots of the Interactor object. For example, you can easily specify which mouse button or keyboard key starts the interactor. In order to affect the graphics and connect to application programs, each Interactor has multiple protocols. For example, the "Move-Grow" interactor, for moving graphical objects with the mouse, explicitly sets the `Am_LEFT` and `Am_TOP` slots of the object, and also calls the `Am_DO_METHOD` method stored in the Command object attached to the Interactor. Therefore, there are multiple ways to use an Interactor, to give programmers flexibility in what they need to achieve.

When an Interactor or a *widget* (see the Widgets chapter) finishes its operation, it allocates a *Command object* and then invokes the 'do' method of that Command object. Thus, the Command objects take the place of *call-back procedures* in other systems. The reason for having Command objects is that in addition to the'do' method, a Command object also has methods to support undo, redo, selective undo and redo, help, and enabling of operations. Each Interactor and Widget has a Command object as the part named `Am_COMMAND`, and Interactors set the `Am_VALUE` and other slots in its command object, and then call the `Am_DO_METHOD` method. This and other methods in the Command objects implement the functionality of the Interactors.

Amulet currently supports two undo models, fully described in Section 5.5.2. The first is a simple single undo, like on the Macintosh. The second is a sophisticated new undo model which provides undo, redo (undo the undo), and selective undo and repeat of any previous command. The sections about the various Interactors discuss their default operation for undo, redo and repeat.

5.3 Standard Operation

We hope that most normal behaviors and operations will be supported by the Interactors and Command objects in the library. This section discusses how to use these. If you find that the standard operations are not sufficient, then you can override the standard methods, as described in Section 5.5.2. If you want an *additional* operation in addition to the regular operation, then you can just add a command object to the Interactor, as explained in Section 5.5. If neither of these is sufficient, you may need to create your own Interactor as discussed in Section 5.7.

5.3.1 Designing Behaviors

The first task when designing the interaction for your interface is to choose the desired behavior. The first choice is whether one of the built-in widgets provides the right interface. If so, then you can choose the widget from the Widgets chapter and then attach the appropriate Command object to the widget. The widgets, such as buttons, scroll bars and text-input fields, combine a standard graphical presentation with an interactive behavior. If you want custom graphics, or you want an application-specific graphical object to be moved, selected or edited with the mouse, then you will want to create your own graphics and Interactors.

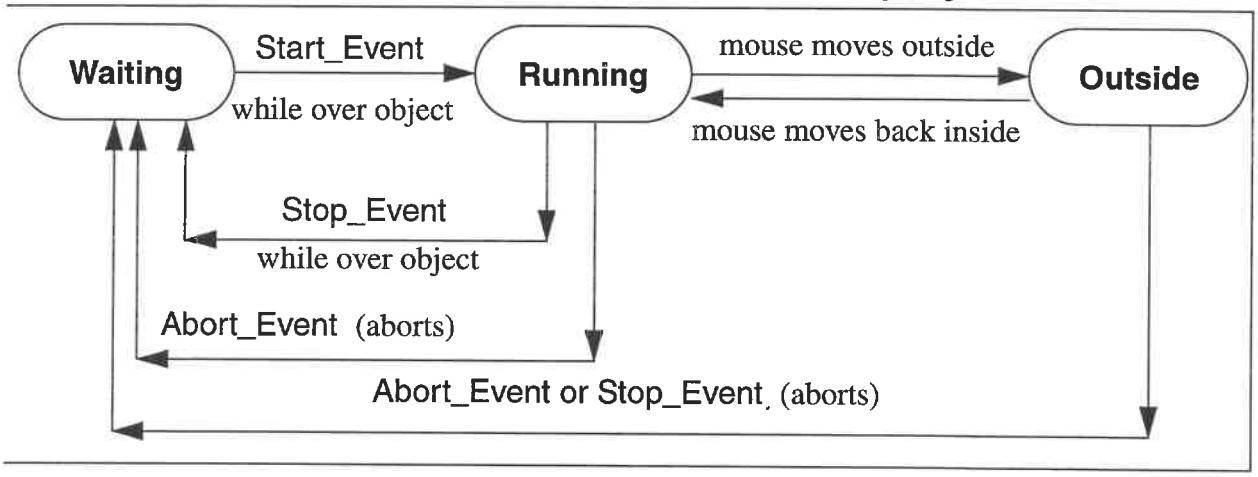
The first step in programming an Interactor is to pick one of the fundamental built-in styles of behavior that is closest to the interaction you want. The current choices are (these are exported in `inter.h`):

- `Am_Choice_Interactor`. This is used to choose one or more from a set of objects. The user is allowed to move around over the objects (getting *interim feedback*) until the correct item is found, and then there will often be *final feedback* to show the final selection. The `Am_Choice_Interactor` can be used for selecting among a set of buttons or menu items, or choosing among the objects dynamically created in a graphics editor.
- `Am_One_Shot_Interactor`. This is used whenever you want something to happen immediately when an event occurs, for example when a mouse button is pressed over an object, or when a particular keyboard key is hit. Like the `Am_Choice_Interactor`, the `Am_One_Shot_Interactor` can be used to select among a set of objects, but the `Am_One_Shot_Interactor` will not provide interim feedback the object where you initially press will be the final selection. The `Am_One_Shot_Interactor` is also useful in situations where you are not selecting an object, such as when you want to get a single keyboard key.
- `Am_Move_Grow_Interactor`. This is useful in all cases where you want a graphical object to be moved or changed size with the mouse. It can be used for dragging the indicator of a scroll bar, or for moving and growing objects in a graphics editor.
- `Am_New_Points_Interactor`. This Interactor is used to enter new points, such as when creating new objects. For example, you might use this to allow the user to drag out a rubber-band rectangle for defining where a new object should go.
- `Am_Text_Edit_Interactor`. This supports editing the text string of a text object. It supports a flexible *key translation table* mechanism so that the programmer can easily modify and add editing functions. The built-in mechanisms support basic text editing behaviors.

- `Am_Gesture_Interactor`. This Interactor supports free-hand gestures, such as drawing an X over an object to delete it, or encircling the set of objects to be selected.
- `Am_Rotate_Interactor`. This Interactor will support rotating graphical objects. It is not yet implemented.
- `Am_Animation_Interactor`. This Interactor will support animations and time-based events. It is not yet implemented.

5.3.2 General Interactor Operation

Once an Interactor is created and its parameters are set (see Section 5.3.3), the programmer will then attach the Interactor to some object in a window (see Section 5.3.3.2.1). Amulet then waits for the user to perform the Interactor's start event (for example by pressing the left mouse button, see Section 5.3.3.1.5) over the graphical object to which the Interactor is attached. `Am_One_Shot_Interactors` then immediately execute their Command's `DO` method and go back to waiting. Other types of Interactors, however, usually show interim feedback while waiting for a specific stop event (for example, left mouse button up). While Interactors are operating, the user might move the mouse outside the Interactor's operating area, in which case the Interactor stops working (for example, a choice Interactor used in a menu will turn off the highlighting if the mouse goes outside the menu). If the mouse goes back inside, then the Interactor resumes operation. If the abort_event is executed while an Interactor is running, then it is aborted (and the Command's `DO` method is not executed). Similarly, if the stop event is executed while the mouse is outside, the Interactor also aborts. The operation is summarized by the following diagram.



Multiple Interactors can be running at the same time. Each Interactor keeps track of its own status, and for each input event, Amulet checks which Interactor or Interactors are interested in the event. The appropriate Interactor(s) will then process that event and return.

5.3.3 Parameters

Once the programmer has chosen the basic behavior that is desired, then the various parameters of the specific Interactor must be filled in. The next sections give the details of these parameters. Some, such as the start and abort events, are shared by all Interactors, and other parameters, such as gridding, are specific to only a few types of Interactors.

The parameters are set as normal slots of the objects. The names of the slots are defined in `standard_slots.h` and are described below. As an example, the following creates a choice Interactor called 'Select It' assigned to the variable `select_it` and sets its start event to the middle mouse button. See the ORE chapter for how to create and name objects.

```
select_it = Am_Choice_Interactor.Create("Select It")
                .Set(Am_START_WHEN, "MIDDLE_DOWN");
```

5.3.3.1 Events

One of the most important parameters for all Interactors are the input events that cause them to start, stop and abort. These are encoded as an `Am_Input_Char` which are defined in `idefs.h`. Normally, you do not have to worry about these since they are automatically created out of normal C strings, but you can convert a string into an `Am_Input_Char` for efficiency, or if you want to set or access specific fields.

Note: Do not use a C++ `char` to represent the events. It must be a C string or an `Am_Input_Char` object.

5.3.3.1.1 Event Slots

There are three slots of Interactors that can hold events: `Am_START_WHEN`, `Am_ABORT_WHEN`, and `Am_STOP_WHEN`.

`Am_START_WHEN` determines when the Interactor begins operating. The default value is `Am_Default_Start_Char` which is `"LEFT_DOWN"` with no modifier keys (see Section 5.3.3.1.3) but with any number of clicks (see Section 5.3.3.1.4). So by default, all interactors will operate on both single and double clicks.

`Am_ABORT_WHEN` allows the Interactor to be aborted by the user while it is operating. The default value is `"CONTROL_g"`. Aborting is different from undoing since you abort an operation *while it is running*, but you undo an operation *after it is completed.* All interactors can be aborted while they are running.

`Am_STOP_WHEN` determines when the Interactor should stop. The default value is `Am_Default_Stop_Char` which is `"ANY_MOUSE_UP"` so even if you change the start_when, you can often leave the stop_when as the default value.

5.3.3.1.2 Event Values

In any of these slots, you can provide an `Am_Input_Char`, a string in the format described below, or the special values `true` or `false`. The value `true` matches any event, and `false` will never match any event. You might use false in the `Am_ABORT_WHEN` slot of an Interactor to make sure it is never aborted.

The general form for the events is a string with the modifiers first and the specific keyboard key or mouse button last. The specific keys include the regular keyboard keys, like `"A"`, `"z"`, `"["`, and `"\""` (use the standard C++ mechanism to get special characters into the string). The various function and special keys are generally named the same thing as their label, such as `"F1"`, `"R5"`, `"HELP"`, and `"DELETE"`. Sometimes, keys have multiple markings, in which case we usually use the more specific or textual marking, or sometimes both markings will work. Also, the arrow keys are always called `"LEFT_ARROW"`, `"UP_ARROW"`, `"DOWN_ARROW"`, and `"RIGHT_ARROW"`. Note that keys with names made out of multiple words are separated by underscores. For keyboard keys, we currently only support operations on the button being pressed, and no events are generated when the button is released. You can specify any keyboard key with the special event `"ANY_KEYBOARD"` (see Section 5.4.1 for how to find out which key was hit). You can find out what the mapping for a keyboard key is by running the test program `testinput`, which is in the `src/gem` directory. We have tried to provide appropriate mappings for all of the keyboards we have come across, but if there are keyboard keys on your keyboard that are not mapped appropriately, then please send mail to `amulet@cs.cmu.edu` and we will add them to the next release.

For the mouse buttons, we support both pressing and releasing. The names of the mouse buttons are `"LEFT"`, `"MIDDLE"` and `"RIGHT"` (on a 2-button mouse, they are `"LEFT"` and `"RIGHT"` and on a 1-button mouse, just `"LEFT"`), and you must append either `"UP"` or `"DOWN"`. Thus, the event for the left button down is `"LEFT_DOWN"`. You can specify any mouse button down or up using `"ANY_MOUSE_DOWN"` and `"ANY_MOUSE_UP"`. On the Macintosh, you can generate the right mouse down event using OPTION-mouse down, and the middle mouse event using OPTION-SHIFT-mouse down. On the PC with a two-button mouse, there is no way to generate the middle button event.

5.3.3.1.3 Event Modifiers

The modifiers can be specified in any order and the case of the modifiers does not matter. There are long and short prefix forms for each modifier. You can use either one in strings to be converted into `Am_Input_Chars`. For example `"CONTROL_f"` and `"^f"` represent the same key. Note that the short form uses a hypen (it looks better in menus) and the long form uses an underscore (to be consistent with other Amulet symbols).

The currently supported modifiers are:

- `shift_` or `SHFT-` One of the keyboard shift keys is being held down. For letters, you can also just use the upper case. Thus, `"F"` is equivalent to `"SHIFT_f"`. However, do not use shift to try to get the special characters. Therefore `"Shift_5"` is *not* the same as `"%"`. For alphabetic characters only, the Caps Lock key produces a `shift` modifier.

- `control_` or `^` The control key is being held down.
- `meta_` or `MET-` The meta key is the diamond key on Sun keyboards, the `EXTEND-CHAR` key on HPs, the Command (Apple) key on Macintosh keyboards, and the `ALT` key on PC keyboards. On other Unix keyboards, it is generally whatever is used for "meta" by Emacs and the window manager. Note that on the Macintosh, the Option key with the mouse button generates middle and right button events, and with keyboard keys, just returns whatever is at that point in the font. Similarly, the default text interactor uses the Meta key on Unix to generate characters at the top of the font (see Section 5.3.5.5.1).
- `any_` This means that you don't care which modifiers are down. Thus `"any_f"` matches `"shift_f"` as well as `"F"` and `"meta_control_shift_f"`. Note that `"ANY_KEYBOARD"` or `"ANY_MOUSE_DOWN"` also specifies any modifiers.

5.3.3.1.4 Multiple Clicks

Amulet supports the detection of multiple click events from the mouse. To double-click, the user must press down on the same mouse button quickly two times in succession. The clicks must be faster than `Am_Double_Click_Time` (which is defined in gem.h), which defaults to 250, and is measured in milliseconds. (On the Macintosh, `Am_Double_Click_Time` is ignored, and the system constant for double click time is used instead, which is set with the Mouse control panel.)

On the PC, Amulet detects single and double clicks, and on Unix and the Mac, Amulet will detect up to five clicks. The multiple clicks are named by preceding the event name with the words `"DOUBLE_"`, `"TRIPLE_"`, `"QUAD_"`, and `"FIVE_"`. For example, `"double_left_down"`, or `"shift_meta_triple_right_down"`. When the user double clicks, a single click event will still be generated first. For example, for the left button, the sequence of received events will be `"LEFT_DOWN"`, `"LEFT_UP"`, `"DOUBLE_LEFT_DOWN"`, `"DOUBLE_LEFT_UP"`. The `"ANY_"` prefix can be used to accept any number of clicks, so `"ANY_LEFT_DOWN"` will accept single or multiple clicks with any modifier held down.

5.3.3.1.5 Am_Input_Char type

The `Am_Input_Char` is defined in `idefs.h`. It is a regular C++ object (*not* an Amulet object). It has constructors from a string or from the various pieces:

```
Am_Input_Char (const char *s); //from a string like "META_LEFT_DOWN"
Am_Input_Char (short c = 0, bool shf = false,
               bool ctrl = false,
               bool meta = false, Am_Button_Down down = Am_NEITHER,
               Am_Click_Count click = Am_NOT_MOUSE,
               bool any_mod = false);
```

It can be converted to a string, to a short string, to a long (which is only useful for storing the `Am_Input_Char` into a slot of an object) or to a character (which returns 0 if it is not an normal ascii character). An `Am_Input_Char` will also print to a stream as a string. If `ic` is an `Am_Input_Char`:

- `ic.As_String(char *s);` convert to a string by writing into `s`, which should be at least `Am_LONGEST_CHAR_STRING` characters long.

- `ic As_Short_String(char *s);` Convert to a short string like "^C" such as might be used in a menu. `s` should be at least `Am_LONGEST_CHAR_STRING` characters long.
- `(long)ic;` convert `ic` into a long, for storing it into a slot.
- `char c = ic.As_Char();` Returns a char if ic represents a simple ascii character, otherwise returns \0.
- `cout << ic;` you can print an `Am_Input_Char` directly, in which case it prints the same as `As_String`.
- `ic = Am_Input_Char::Narrow (obj.Get(SLOT));` can be used to convert a slot holding a `long` into an `Am_Input_Char`.

The member variables of an `Am_Input_Char` are:

```
typedef enum { Am_NOT_MOUSE = 0,   //When not a mouse button.
               Am_SINGLE_CLICK = 1, //Also for mouse moved, with Am_NEITHER.
               Am_DOUBLE_CLICK = 2, Am_TRIPLE_CLICK = 3,
               Am_QUAD_CLICK = 4, Am_FIVE_CLICK = 5, Am_MANY_CLICK = 6,
               Am_ANY_CLICK = 7   // when don't care about how many clicks
             } Am_Click_Count;

typedef enum { Am_NEITHER = 0, Am_BUTTON_DOWN = 1,
               Am_BUTTON_UP = 2, Am_ANY_DOWN_UP = 3} Am_Button_Down;

short code; // the base code.
bool shift;  // whether these modifier keys were down
bool control;
bool meta;
bool any_modifier; //true if don't care about modifiers
Am_Button_Down button_down;   // whether a down or up transition.
                              // For keyboard, only support down.
Am_Click_Count click_count;   // 0==not mouse, otherwise # clicks
```

5.3.3.2 Graphical Objects

5.3.3.2.1 Start_Where

For an Interactor to become active, it must be added as a part to a graphical object which is part of a window. To do this, you use the regular Add_Part method of objects. For example, to make the `select_it` Interactor defined above in Section 5.3.3 select the object `my_rect`, the following code could be used:

```
my_rect.Add_Part(select_it);
```

Interactors can be added as parts to any kind of graphical object, including primitives (like rectangles and strings), groups, and windows. You can add multiple Interactors to any object, and they can be interspersed with graphical parts for groups and windows. Interactors can be removed or queried with the standard object routines for parts. If you make instances of the object to which the Interactor is attached, then an instance will be made of the Interactor as well (see the ORE chapter). For example:

```
Am_Slot_Key INTER_SLOT = Am_Register_Slot_Name ("INTER_SLOT");
my_rect.Add_Part(INTER_SLOT, select_it); //named part
```

```
rect2 = my_rect.Create(); //rect2 will have its own which is an instance of select_it
```

It is very common for a behavior to operate over the *parts* of a group, rather than just on the object itself. For example, a choice Interactor might choose any of the items (parts) in a menu (group), or a move_grow Interactor might move any of the objects in the graphics window. Therefore, the slot `Am_START_WHERE_TEST` can hold a function to determine where the mouse should be when the start-when event happens for the Interactor to start. The built-in functions for the slot (from `inter.h`) are as follows. Each of these returns the object over which the Interactor should start, or NULL if the mouse is in the wrong place so the Interactor should not start.

- `Am_Inter_In_Object_Or_Part`: If the interactor is attached to a group-like object (a `Am_Window`, `Am_Screen`, `Am_Group` or `Am_Scrolling_Group`), then looks for a part of that object for the mouse to be in. Otherwise, tests whether the mouse is directly in the object the mouse is attached to. This is the default `Am_START_WHERE_TEST` for most interactors.
- `Am_Inter_In_Text_Object_Or_Part`: If the interactor is attached to a group-like object (a `Am_Window`, `Am_Screen`, `Am_Group` or `Am_Scrolling_Group`), then looks for a part of that object of type `Am_Text` for the mouse to be in. Otherwise, tests whether the mouse is directly in the object the mouse is attached to and that object is a `Am_Text`. This is the default `Am_START_WHERE_TEST` for `Am_Text_Edit_Interactors`.
- `Am_Inter_In`: If the mouse is inside the object the Interactor is part of, this returns that object.
- `Am_Inter_In_Part`: The Interactor should be part of a group or window object. This tests if the mouse is in a part of that group or window object, and if so, returns the part of the group or window the mouse is over.
- `Am_Inter_In_Leaf`: This is useful when the Interactor is part of a group or window which contains groups which contain groups, etc. It returns the lowest level object the mouse is over. If you want `Am_Inter_In_Leaf` to return a group rather than a part of the group, set the `Am_PRETEND_TO_BE_LEAF` slot of the group to be `true`.
- `Am_Inter_In_Text`: If the mouse is inside the object the Interactor is part of, and that object is an instance of `Am_Text`, then returns that object. This is useful for `Am_Text_Interactors`.
- `Am_Inter_In_Text_Part`: If the mouse is in a part of the object the Interactor is part of, and that the part the mouse is over is an instance of `Am_Text`, then returns that part. This is useful for `Am_Text_Interactors`.
- `Am_Inter_In_Text_Leaf`: If the mouse is in a leaf of the object the Interactor is part of, and that leaf part is an instance of `Am_Text`. This is useful for `Am_Text_Interactors`.

For example, the following interactor will move whichever part of the group `my_group` that the user clicks on. Since the interactor is also a part of the group, an instance of the interactor will be created whenever an instance is made of the group, as explained in Section 3.6.2.

```
Am_Slot_Key INTER_SLOT = Am_Register_Slot_Name ("INTER_SLOT");
my_group.Add_Part(INTER_SLOT, Am_Move_Grow_Interactor.Create()
                  .Set(Am_START_WHERE_TEST, Am_Inter_In_Part));
my_group.Add(rect);
my_group.Add(rect2);
//now the interactor will move either rect or rect2
group2 = my_group.Create();
```

```
//group2 will have its own interactor as well as instances of rect and rect2
```

If none of these functions returns the object you are interested in, then you are free to define your own function. It should be a method of type `Am_Where_Method`, and should return the object the Interactor should manipulate, or `Am_No_Object` if none. For example:

```
Am_Define_Method(Am_Where_Method, Am_Object, in_special_obj_part,
          (Am_Object /* inter */,
           Am_Object object, Am_Object event_window,
           Am_Input_Char /*ic*/, int x, int y)) {
  Am_Object val = Am_Point_In_Part(object, x, y, event_window);
  if (val.Valid() && (bool)val.Get(MY_SPECIAL_SLOT)) return val;
  else return Am_No_Object;
}
```

Note that this means that the Interactor may actually operate on an object *different* from the one to which it is attached. For example, Interactors will often be attached to a group but actually modify a part of that group. With a custom `Am_START_WHERE_TEST` function, the programmer can have the Interactor operate on a completely independent object.

5.3.3.3 Active

It is often convenient to be able to create a number of Interactors, and then have them turn on and off based on the global mode or application state. The `Am_ACTIVE` slot of an Interactor can be set to `false` to disable the Interactor, and it can be set to `true` to re-enable the Interactor. By default, all Interactors are active. Setting the `Am_ACTIVE` slot is more efficient than creating and destroying the Interactor. The `Am_ACTIVE` slot can also be set with a constraint that returns `true` or `false`.

5.3.3.4 Am_Inter_Location

Some interactors require a parameter to describe a location and/or size of an object. Since the coordinate system of each object is defined by its group, just giving a number for X and Y would be meaningless without also supplying a reference object. Therefore, we have introduced a wrapper type, called `Am_Inter_Location`, which encapsulates the coordinates with a reference object. As described in the Opal manual, some objects are defined by their left, top, width and height, and others by two end points, and this information is also included in the `Am_Inter_Location`. The methods on an `Am_Inter_Location` (from `inter.h`) are:

```
class Am_Inter_Location {
public:
  Am_Inter_Location (); // empty

  //create a new one. If as_line, then a,b is x1, y1 and c,d is x2, y2.
  // If not as_line, then a,b is left, top and c,d is width, height
  Am_Inter_Location (bool as_line, Am_Object ref_obj,
                     int a, int b, int c, int d);

  //change the values of an existing one
  void Set_Location (bool as_line, Am_Object ref_obj,
                     int a, int b, int c, int d, bool make_unique = true);

  //change just the first coordinate of an existing one
```

```
    void Set_Location (bool as_line, Am_Object ref_obj,
                       int a, int b, bool make_unique = true);

    //return all the values
    void Get_Location (bool &as_line, Am_Object &ref_obj,
                       int &a, int &b, int &c, int &d) const;

    //return just the coordinates, the reference object, whether it is a line or not
    void Get_Points (int &a, int &b, int &c, int &d) const;
    Am_Object Get_Ref_Obj () const;
    void Get_As_Line (bool &as_line) const;

    //copy from or swap with another Am_Inter_Location
    void Copy_From (Am_Inter_Location& other_obj, bool make_unique = true);
    void Swap_With (Am_Inter_Location& other_obj, bool make_unique = true);

    //translate the coordinates so they now are with respect to dest_obj
    bool Translate_To(Am_Object dest_obj);

    Am_Inter_Location Copy() const; //make a new one like me
    virtual void Print_Name (ostream& os); //print my contents on the stream
};
```

5.3.4 Top Level Interactor

Am_Interactor is used to build new, custom interactors. This object won't do anything if you simply instantiate it and add it to a window.

Slot	Default Value	Type	
Am_START_WHEN	Am_Default_ Start_Char	Am_Input_Char	
Am_START_WHERE_TEST	Am_Inter_In_ Object_Or_Part	Am_Where_ Method	
Am_ABORT_WHEN	Am_Input_Char ("CONTROL_g")	Am_Input_Char	
Am_INTER_BEEP_ON_ABORT	true	bool	
Am_RUNNING_WHERE_OBJECT	true	Am_Object, bool	
Am_RUNNING_WHERE_TEST	Am_Inter_In_ Object_Or_Part	Am_Where_ Method	
Am_STOP_WHEN	Am_Default_ Stop_Char	Am_Input_Char	
Am_ACTIVE	true	bool	*Section 5.3.3.3*
Am_START_OBJECT	0	Am_Object	*Section 5.4.1*
Am_START_CHAR	0	Am_Input_Char	*Section 5.4.1*
Am_CURRENT_OBJECT	0	Am_Object	*Section 5.4.1*
Am_RUN_ALSO	false	bool	*Section 5.4.2*
Am_PRIORITY	1.0	float	*Section 5.4.2*
Am_MULTI_OWNERS	NULL	Am_Value_List or NULL	*Section 5.4.3*
Am_MULTI_FEEDBACK_ OWNERS	NULL	Am_Value_List or NULL	*Section 5.4.3*
Am_WINDOW	NULL	Am_Window	*Set with current window*
Am_COMMAND	Am_Command	Am_Command	*Section 5.6*

These are the default values of Am_Interactor's slots. Most of these are advanced features and are discussed in Section 5.4.

5.3.5 Specific Interactors

All of the interactors and command objects are summarized in Chapter 10, *Summary of Exported Objects and Slots*. The next sections discuss each one in detail.

5.3.5.1 Am_Choice_Interactor

The Am_Choice_Interactor is used whenever the programmer wants to choose one or more out of a set of objects, such as in a menu or to select objects in a graphics window. The standard behavior allows the programmer to choose whether one or more objects can be selected, and special slots called Am_INTERIM_SELECTED and Am_SELECTED of these objects are set by default. Typically, the programmer would define constraints on the look of the object (e.g. the color) based on the values of these slots. Note that Am_INTERIM_SELECTED and Am_SELECTED are set in the *graphical object* the Interactor operates on, not in the Interactor itself.

Slot	Default Value	Type	
All of ths slots of Am_Interactor, with the following changes:			
Am_RUNNING_WHERE_OBJECT	<formula>	Am_Object, bool	*computes owner*
Am_RUNNING_WHERE_TEST	<formula>	Am_Where_Method	*same as start*
Am_HOW_SET	Am_CHOICE_TOGGLE	Am_Choice_How_Set	
Am_FIRST_ONE_ONLY	false	bool	*whether menu- or button-like*
Am_VALUE	NULL	Am_Object or Am_Value_List of objects	

5.3.5.1.1 Special Slots of Choice Interactors

Two slots of choice Interactors can be set to customize its behavior:

- Am_HOW_SET: This controls whether a single or multiple values will be selected. Legal values are from the following type:

```
typedef enum (Am_CHOICE_SET, Am_CHOICE_CLEAR, Am_CHOICE_TOGGLE,
              Am_CHOICE_LIST_TOGGLE } Am_Choice_How_Set;
```

These mean:

- Am_CHOICE_SET: the object under the mouse becomes selected, and the previously selected object is de-selected (useful for single selection menus and radio buttons). Unlike Am_CHOICE_TOGGLE, clicking on an already-selected object leaves it selected.
- Am_CHOICE_CLEAR: the object under the mouse becomes de-selected. This is rarely useful.
- Am_CHOICE_TOGGLE: if the object under the mouse is selected, it becomes deselected, otherwise it becomes selected and any previous object become de-selected. This is useful when you want zero or one selection (the user is able to turn off the selection).

- Am_CHOICE_LIST_TOGGLE: if the object under the mouse is selected, then it is deselected, otherwise it becomes selected, but other objects are left alone. This allows multiple selection, and is useful for check boxes.

The default value for the Am_HOW_SET slot is Am_CHOICE_TOGGLE.

- Am_FIRST_ONE_ONLY: If false (the default), then the selection is free to move from one item in the group to another, as in menus. If true, then only the initial object the mouse is over can be manipulated, and the user must release outside and then press down in another object to change objects. This is how radio button and check box widgets work on most systems.

5.3.5.1.2 Standard operation of the Am_Choice_Interactor

As the Choice_Interactor is operating, it calls the various internal methods. The default operation of these methods is as follows. If this is not sufficient for your needs, then you may need to override the methods, as explained in Section 5.5.2.

As the Interactor moves over various graphical objects, the Am_INTERIM_SELECTED slot of the object is set to true for the object which is under the mouse, and false for all other objects. Typically, the graphical objects that the Interactor affects will have a constraint to the Am_INTERIM_SELECTED slot from the Am_FILL_STYLE or other slot. At any time, the Interactor can be aborted by typing the key in Am_ABORT_WHEN (the default is "control_g"). When the Am_STOP_WHEN event occurs, the Am_INTERIM_SELECTED slot is set to false, and the Am_HOW_SET slot of the Interactor is used to decide how many objects are allowed to be selected (as explained above). The objects that should end up being selected have their Am_SELECTED slot set to true, and the rest of the objects have their Am_SELECTED slot set to false. Also the Am_VALUE slot of the Interactor and the Am_VALUE slot of the command object in the Am_COMMAND slot of the Interactor will contain the current value. If Am_HOW_SET is not Am_CHOICE_LIST_ TOGGLE, then the Am_VALUE slot will either contain the selected object or Am_No_Object (NULL). If Am_HOW_SET is Am_CHOICE_LIST_TOGGLE, then the Am_VALUE slot of the Command object will contain a Am_Value_List containing the list of the selected objects (or it will be the empty list).

The default undo of the Am_Choice_Interactor simply resets the Am_SELECTED slots of the selected object(s) and the Am_Value of the Interactor and the command object to be as they were before the Am_Choice_Interactor was run. Redo restores the values.

5.3.5.1.3 Simple Example

See the file testinter.cc for lots of additional examples of uses of Interactors and Command objects. The following Interactor works on any object which is directly a part of the window. Due to the constraints, if you press the mouse down over any rectangle created from rect_proto that is in the window, it will change to having a thick line style when they mouse is over it (when it is "interim-selected"), and they will turn white when the mouse button is release (and it becomes selected).

```
Am_Define_Style_Formula (rect_line)  {
  if ((bool)self.GV (Am_INTERIM_SELECTED)) return thick_line;
  else return thin_line;
}
```

```
Am_Define_Style_Formula (rect_fill)  {
  if ((bool)self.GV (Am_SELECTED)) return Am_White;
  else return self.GV (Am_VALUE);   //the real color
}
rect_proto = Am_Rectangle.Create ("rect_proto")
   .Set (Am_WIDTH, 30)
   .Set (Am_HEIGHT, 30)
   .Set (Am_SELECTED, false)
   .Set (Am_INTERIM_SELECTED, false)
   .Set (Am_VALUE, Am_Purple)   //put the real color here
   .Set (Am_FILL_STYLE, rect_fill)
   .Set (Am_LINE_STYLE, rect_line)
   ;
select_inter = Am_Choice_Interactor.Create("choose_rect")
               .Set (Am_START_WHERE_TEST, Am_Inter_In_Part);
window.Add_Part (select_inter);
```

5.3.5.2 Am_One_Shot_Interactor

The Am_One_Shot_Interactor is used when you want something to happen immediately on an event. For example, you might want a command to be executed when a keyboard key is hit, or when the mouse button is first pressed. The parameters and default behavior for the Am_One_Shot_Interactor are the same as for a Am_Choice_Interactor, in case you want to have an object be selected when the start_when event happens. The programmer can choose whether one or more objects can be selected, and the slots Am_INTERIM_SELECTED and Am_SELECTED of these objects are set by the Am_One_Shot_Interactor the same was as the Am_Choice_Interactor.

The slots for the Am_One_Shot_Interactor are identical to those for the Am_Choice_Interactor (see above).

5.3.5.2.1 Simple Example

In this example, we create a Am_One_Shot_Interactor which calls the Am_DO_METHOD of the change_setting_command (which is do_change_setting) when any keyboard key is hit in the window. The change_setting_command's Am_UNDO_METHOD (which is undo_change_setting) will be used to undo this action. The programmer would write the methods for do and undo.

```
Am_Object change_setting_command = Am_Command.Create()
         .Set(Am_DO_METHOD, do_change_setting)
         .Set(Am_UNDO_METHOD, undo_change_setting);

Am_Object how_set_inter =
  Am_One_Shot_Interactor.Create("change_settings")
         .Set(Am_START_WHEN, "ANY_KEYBOARD")
         .Add_Part(Am_COMMAND, change_setting_command)
         ;
window.Add_Part (how_set_inter);
```

5.3.5.3 Am_Move_Grow_Interactor

The Am_Move_Grow_Interactor is used to move or change the size of graphical objects with the mouse. The default methods in the Am_Move_Grow_Interactor directly set the appropriate slots of the object to cause it to move or change size. For rectangles, circles, groups and most other objects, the default methods set the Am_LEFT, Am_TOP, Am_WIDTH and Am_HEIGHT. For lines (more specifically, any object whose Am_AS_LINE slot is true), the methods may instead set the Am_X1, Am_Y1, Am_X2 and Am_Y2 slots.

Slot	Default Value	Type	
All of ths slots of Am_Interactor, with the following changes:			
Am_GROWING	false	bool	
Am_AS_LINE	<formula>	bool	
Am_FEEDBACK_OBJECT	NULL	Am_Object	*interim feedback*
Am_GRID_X	0	int	
Am_GRID_Y	0	int	
Am_GRID_ORIGIN_X	0	int	
Am_GRID_ORIGIN_Y	0	int	
Am_GRID_METHOD	NULL	Am_Custom_ Gridding_Method	
Am_WHERE_ATTACH	Am_ATTACH_ WHERE_HIT	Am_Move_Grow_ Where_Attach	*Am_ATTACH_.. {WHERE_HIT, NW, N, NE, E, SE, S, SW, W, END_1, END_2, CENTER}*
Am_MINIMUM_WIDTH	0	int	
Am_MINIMUM_HEIGHT	0	int	
Am_MINIMUM_LENGTH	0	int	
Am_VALUE	NULL	Am_Inter_Location	

5.3.5.3.1 Special Slots of Move_Grow Interactors

- Am_GROWING: If false or zero, then object is moved without changing its size (or for lines, without changing the orientation or length). If true or non-zero, then adjusts the size (or a single end-point for a line). The default is false.
- Am_AS_LINE: If false or zero, then treats the object as a rectangle and adjusts the Am_LEFT, Am_TOP, Am_WIDTH and Am_HEIGHT slots. If true or non-zero, then if the object is being changed size, then sets the Am_X1, Am_Y1, Am_X2 and Am_Y2 slots (lines can be moved by setting their Am_LEFT and Am_TOP slots). The default is a formula that looks at the value of the Am_AS_LINE slot of the object the Interactor is modifying.
- Am_FEEDBACK_OBJECT: If NULL (0) (the default), then the actual object moves around with the mouse. If this slot contains an object, however, then that object is used as an interim-feedback object, and it moves around with the mouse, and the actual object is moved or changed size only when the stop-when event happens (e.g., when the mouse button is released). *Don't forget to add the feedback object to a group or window* ***in addition*** *to*

adding it as the Am_FEEDBACK_OBJECT. While the feedback object is moving around, the original object simply stays in its original position. See Section 5.4.3 about using a window as the feedback object.

- Am_WHERE_ATTACH: This slot controls what part of the object is attached to the mouse as the object is manipulated. The options are defined by the enum type Am_Move_Grow_Where_Attach in inter.h. They are:
  - Am_ATTACH_WHERE_HIT: (This is the default.) The mouse is attached where the mouse is pressed down. If growing the object, then checks which edge the mouse is closest to, and grows from there.
  - Am_ATTACH_CENTER: The center of the object. This is illegal if growing the object.
  - Am_ATTACH_NW, Am_ATTACH_N, Am_ATTACH_NE, Am_ATTACH_E,Am_ATTACH_SE, Am_ATTACH_S, Am_ATTACH_SW, Am_ATTACH_W: The mouse is attached at this corner or at the center of this side of the object.
  - Am_ATTACH_END_1, Am_ATTACH_END_2: Only available for lines. End_1 is the end defined by Am_X1 and Am_Y1.
- Am_MINIMUM_WIDTH, Am_MINIMUM_HEIGHT : When growing, these are the minimum legal size. Default is 0.
- Am_MINIMUM_LENGTH: Minimum length when growing lines. Default is 0.

5.3.5.3.2 Gridding

There are two ways to do gridding for Am_Move_Grow_Interactors and Am_New_Point_Interactors. The first is to provide a method, and the second is the provide the gridding origin and multiples:

- Am_GRID_METHOD: (Default is NULL (0)). If supplied, this should be a method of the type Am_Custom_Gridding_Method. This function will be given the current x and y and should return the new x and y to use. This kind of gridding is also useful for snapping, "gravity," and keeping the object being dragged inside a region. For example:

```
Am_Define_Method(Am_Custom_Gridding_Method, void, keep_inside_window,
      (Am_Object inter, int x, int y,
           int& out_x, int & out_y)) { ... }
```

//see the file samples/space/space.cc *for the complete code of this function*

- Am_GRID_X, Am_GRID_Y: If Am_GRID_PROC is not supplied, then these slots can hold the number of pixels the mouse skips over. Default is 0.
- Am_GRID_ORIGIN_X, Am_GRID_ORIGIN_Y: These can hold the offset in pixels from the edge of the window for the origin of the gridding. Default is 0.

5.3.5.3.3 Standard operation of the Am_Move_Grow_Interactor

As the Am_Move_Grow_Interactor is operating, it calls the various internal methods. The default operation of these methods is as follows. If this is not sufficient for your needs, then you may need to override the methods, as explained in Section 5.5.2.

If the Interactor's Am_GROWING slot is set to true, the interactor grows the object, otherwise the interactor moves the object. If the Interactor's Am_AS_LINE slot is false, the object is moved or grown by setting its Am_LEFT, Am_TOP, Am_WIDTH and Am_HEIGHT slots. If the Interactor's Am_AS_LINE slot is true, the object is moved or grown by setting its Am_X1, Am_Y1, Am_X2 and Am_Y2 slots. If there is a feedback object in the Am_FEEDBACK_OBJECT slot then its size is set to the size of the object being manipulated, and its Am_VISIBLE slot is set to true. Then it is moved or its size is changed with the mouse. Otherwise, the object itself is manipulated. At any time while the Interactor is running, the abort event can be hit (default is "control-g") to restore the object to its original position and size. When the stop event happens, then the feedback object is made invisible, and the object is moved or changed size to the final position.

The default Undo method of the Am_Move_Grow_Interactor simply resets the object to its original size and position. Redo undoes the undo. When selectively repeating the operation, there are two possible interpretations: change the object by the same *absolute* values, or change by the same *relative* amounts. We chose to use the absolute values, so repeating the move or grow will put the object in the same place the original object was moved or grown to. Similarly, selective undo will always return the object to where the original object was before the move or grow.

5.3.5.3.4 Simple Example

See the file testinter.cc for additional examples that use Interactors and Command objects. The following Interactor will move any object in the window when the middle button is held down.

```
Am_Object move_inter = Am_Move_Grow_Interactor.Create("move_object")
    .Set (Am_START_WHERE_TEST, Am_Inter_In_Part)
    .Set (Am_START_WHEN, "MIDDLE_DOWN");
window.Add_Part (move_inter);
```

5.3.5.4 Am_New_Points_Interactor

The Am_New_Points_Interactor is used for creating new objects. The programmer can specify how many points are used to define the object (currently, only 1 or 2 points are supported), and the Interactor lets the user rubber-band out the new points. It is generally required for the programmer to provide a feedback object for a Am_New_Points_Interactor so the user can see where the new object will be. If one point is desired, the feedback will still follow the mouse until the stop event, but the final point will be returned, rather than the initial point. Gridding can be used as with a Am_Move_Grow_Interactor. To create the actual new objects, the programmer provides a call-back function in the Am_CREATE_NEW_OBJECT_METHOD slot of the Interactor.

Slot	Default Value	Type	
All of ths slots of Am_Interactor, with the following changes:			
Am_AS_LINE	0	bool	
Am_FEEDBACK_OBJECT	NULL	Am_Object	
Am_HOW_MANY_POINTS	2	int	
Am_FLIP_IF_CHANGE_SIDES	true	bool	
Am_ABORT_IF_TOO_SMALL	false	bool	
Am_GRID_X	0	int	
Am_GRID_Y	0	int	
Am_GRID_ORIGIN_X	0	int	
Am_GRID_ORIGIN_Y	0	int	
Am_GRID_METHOD	0	Am_Custom_ Gridding_Method	
Am_MINIMUM_WIDTH	0	int	
Am_MINIMUM_HEIGHT	0	int	
Am_MINIMUM_LENGTH	0	int	
Am_CREATE_NEW_OBJECT_ METHOD	NULL	Am_Create_New_ Object_Method	
Am_START_WHERE_TEST	Am_Inter_In	Am_Where_Method	
Am_VALUE	NULL	Am_Object	*newly created object*

5.3.5.4.1 Special Slots of Am_New_Points_Interactors

- Am_AS_LINE: If true, then creates the new object as a line, and sets the Am_X1, Am_Y1, etc. slots of the feedback object. If false, the default, then creates the new object as a rectangle, and sets the Am_LEFT, Am_TOP, etc. of the feedback object.
- Am_FEEDBACK_OBJECT: Object to rubber band to show where the new object will be.
- Am_HOW_MANY_POINTS: The number of points that are desired. Lines, rectangles, etc. are normally defined by two points, which is the default. Currently, the only supported values are 1 and 2.

- Am_MINIMUM_WIDTH, Am_MINIMUM_HEIGHT, Am_MINIMUM_LENGTH: Same as for Am_Move_Grow_Interactor (Section 5.3.5.3.1).
- Am_ABORT_IF_TOO_SMALL: If true, and if the size is less than the minimum, then no object will be created (if the stop event happens while the object is less than the minimum, then the Interactor aborts). If false (the default), then an object is created with the minimum size.
- Am_FLIP_IF_CHANGE_SIDES: If true, then if the cursor goes above and/or to the left of the original point, the object is flipped. If false, then the new object is pegged at its minimum size. This is only relevant if Am_AS_LINE is false.
- Am_GRID_X, Am_GRID_Y, Am_GRID_ORIGIN_X, Am_GRID_ORIGIN_Y, Am_GRID_PROC: Same as for Am_Move_Grow_Interactor (Section 5.3.5.3.2).
- Am_CREATE_NEW_OBJECT_METHOD: Set with a method to create the object; see next section.

5.3.5.4.2 Standard operation of the Am_New_Point_Interactor

As the Am_New_Point_Interactor is operating, it calls the various internal methods. The default operation of these methods is as follows. If this is not sufficient for your needs, then you may need to override the methods, as explained in Section 5.5.2.

While the Interactor is operating, the appropriate slots of the feedback object are set, as controlled by the parameters described above. If the user hits the abort key while the Interactor is running ("control_g" by default), the feedback object is made invisible and the Interactor aborts. If the user performs the Am_STOP_WHEN event (usually by releasing the mouse button), then the Am_CREATE_NEW_OBJECT_METHOD is called. (If there is no procedure, then nothing happens.) The method must be of type Am_Create_New_Object_Method which is defined as:

```
// type of method in the Am_CREATE_NEW_OBJECT_METHOD slot of Am_New_Points_Interactor.
// Should return the new object created.
// ** old_object is Valid if this is being called as a result of a Repeat undo call, and means that a new
// object should be created like that old_object.
Am_Define_Method_Type(Am_Create_New_Object_Method, Am_Object,
                      (Am_Object inter, Am_Inter_Location location,
                       Am_Object old_object));
```

The Am_Inter_Location type is explained in Section 5.3.3.4. (Note: this interface may change when we support more than 2 points). After creating the new object and adding it as a part to some group or window, the procedure should return the new object.

The default undo and selective undo methods remove the object from its owner, and the default redo method adds it back to the owner again. The default repeat method calls the Am_CREATE_NEW_OBJECT_METHOD again passing a copy of the original object, and the method is expected to make a new object like the old one. If the create has been undone, then when redo or repeat is no longer possible (determined by the type of undo handler in use--Section 5.5), and the saved objects are automatically destroyed.

Note: if you use the Am_Create_New_Object_Method for something other than creating objects, then do not have a Am_Create_New_Object_Method return the affected object, because the buily-in undo methods may automatically delete the object. For example, in space.cc, a Am_New_Points_Interactor is used to draw the phaser which deletes objects, and this is handled in the command's DO method. It would be an error to do this from the Am_Create_New_Object_Method since the Undo method might delete the object.

5.3.5.5 Am_Text_Edit_Interactor

The Am_Text_Edit_Interactor is used for single-line, single-font editing of the text in Am_Text objects. (Support for multi-line, multi-font text editing will be in a future release.) The default behavior is to directly set the Am_TEXT and Am_CURSOR_INDEX slots of the Am_Text object to reflect the user's changes. Most of the special operations and types used by the Am_Text_Edit_Interactor are defined in text_fns.h.

Slot	Default Value	Type
All of ths slots of Am_Interactor, with the following changes:		
Am_START_WHERE_TEST	Am_Inter_In_Text_ Object_Or_Part	Am_Where_Method
Am_STOP_WHEN	Am_Input_Char("RETURN")	Am_Input_Char
Am_VALUE	""	Am_String
Am_WANT_PENDING_ DELETE	false	bool
Am_TEXT_EDIT_METHOD	Am_Default_Text_ Edit_Method	Am_Text_Edit_ Method
Am_EDIT_TRANSLATION_ TABLE	Am_Edit_Translation_ Table::Default_Table()	Am_Edit_ Translation_Table

When the Am_START_WHEN event occurs, the Interactor puts the text object's cursor where the start event occurred. Subsequent events are sent to the editing method in the Am_TEXT_EDIT_METHOD slot which modifies the Am_Text object. When the stop_when event happens, the cursor is turned off and the command object's DO_METHOD method is called. The stop_when event is *not* entered into the string.

The default Am_RUNNING_WHERE_OBJECT is true, meaning the Interactor will run no matter where the user moves the cursor. If this slot is set to be a particular object, leaving that object causes the Interactor to hide the text object's cursor until the user moves back into the object.

Notice that Am_Text_Edit_Interactor's Am_START_WHERE_TEST slot is set to the value Am_Inter_In_Text_Object_Or_Part. Am_Text_Edit_Interactors only work properly on Am_Text objects, so one of the text tests should be used for the Am_START_WHERE_TEST slot.

5.3.5.5.1 Special Slots of Text Edit Interactors

- Am_WANT_PENDING_DELETE: If this is true (the default is false), then if the user double-clicks in the string, then the entire string is selected. The next character to be typed (unless it is a cursor movement) will delete the entire string.
- Am_TEXT_EDIT_METHOD: This is a method of type Am_Text_Edit_Method (in inter.h) which is defined as:

```
Am_Define_Method_Type(Am_Text_Edit_Method, void,
                      (Am_Object text, Am_Input_Char ic, Am_Object inter));
```

The text edit function should edit the text object's Am_TEXT field given the input character ic. It can also modify the Am_CURSOR_INDEX slot of the object, but shouldn't change other slots. The default function: Am_Default_Text_Edit_Method, uses the Am_Edit_Translation_Table specified in the Interactor's Am_EDIT_TRANSLATION_TABLE slot to provide the basic editing operations (see the description of the Am_EDIT_TRANSLATION_TABLE slot below). If the input character doesn't match any operation in the translation table, the default edit function does the following:

  - ASCII characters between ' ' (SPACE) and '~' are inserted into the string before the cursor.
  - If the META key is held down while typing a character, then the high (eighth) bit of the character is set, so that you can access symbols at the top of the fonts.
  - All other keyboard events make the Interactor beep and do nothing.
  - Non-keyboard events are ignored.

- Am_EDIT_TRANSLATION_TABLE: This is an Am_Edit_Translation_Table (defined in the file text_fns.h), a table that maps input characters to Am_Text_Edit_Operations. Am_Edit_Translation_Table::Default_Table() defines the following mappings:
  - CONTROL_h, BACKSPACE, DELETE: delete character before cursor
  - CONTROL_w, CONTROL_BACKSPACE, CONTROL_DELETE: delete word before cursor
  - CONTROL_d: delete character after cursor
  - CONTROL_u: delete the entire string
  - CONTROL_k: delete string from cursor to end of line
  - CONTROL_b, LEFT_ARROW: move cursor one character to the left
  - CONTROL_f, RIGHT_ARROW: move cursor one character to the right
  - CONTROL_a: move cursor to beginning of line
  - CONTROL_e: move cursor to end of line
  - CONTROL_y: insert the contents of the X cut buffer at the cursor position
  - CONTROL_c: copy the current string into the X cut buffer
  - left mouse button inside the string: move the cursor
  - middle mouse button inside the string: paste X cut buffer
  - any mouse button down outside the string: stop the interactor, as if the user had typed the Am_STOP_WHEN character, but pass the mouse down event on so it can do other

actions as well.

5.3.5.5.2 Standard operation of the `Am_Edit_Text_Interactor`

The text interactor's start action saves a copy of the text object's original `Am_TEXT` slot in the `Am_OLD_VALUE` slot of the Interactor, and it moves the cursor to the location specified by where the Interactor start event occurred. All subsequent events are passed to the `Am_Text_Edit_Method` specified in the Interactor's `Am_TEXT_EDIT_METHOD` slot. An abort event causes the original text object's text to be restored, and the object's `Am_CURSOR_INDEX` is set to `Am_NO_CURSOR`. When the stop event occurs, the Command object's `Am_VALUE` slot is set to the new value of the text object's `Am_TEXT` slot, and the text object's `Am_CURSOR_INDEX` is set to `Am_NO_CURSOR`. The stop_when event is *not* entered into the string.

Undo and selective undo restore the text object to its previous value, and redo undoes the undo. Repeat sets the text object to have the string that the user edited it to.

5.3.5.6 Am_Gesture_Interactor

The `Am_Gesture_Interactor` records a *gesture*, which is the path traced out by the mouse (or other pointing device) while the interactor is running. Beginning with its start event, the gesture interactor records the location of every input event until the interactor stops or aborts. When the interactor stops, it saves the gesture as a list of points (a `Am_Point_List`) in its `Am_POINT_LIST` slot. This behavior (which is the default) is also useful for a free-hand drawing tool, which would retrieve the point list and install it in a `Am_Polygon` object to create a curve duplicating the mouse path.

The real power of the gesture interactor, however, lies in its ability to recognize and classify gestures into categories defined by the programmer. The categories are defined by a `Am_Gesture_Classifier` object installed in the gesture interactor's `Am_CLASSIFIER` slot. (The procedure for creating a classifier is explained in Section 5.3.5.6.1) When a classifier is installed, the gesture interactor attempts to classify each gesture into one of the categories. If a gesture is successfully recognized, the name of its category (a `Am_String`) is stored in the `Am_VALUE` slots of the interactor and its command object. If a gesture is unrecognized -- that is, if it is too different from the prototypical gestures in each category -- then `Am_VALUE` is set to 0.

In addition, to simplify applications where gestures represent commands (such as cut, copy, or paste), the gesture interactor can take a list of command objects in its `Am_ITEMS` slot. Before the interactor invokes its command object, it first searches the `Am_ITEMS` list for a command object whose `Am_LABEL` is identical to `Am_VALUE` (which is the gesture's name if it was recognized and NULL if not). The first matching command object in the list is invoked instead of the interactor's command object. If no command object in `Am_ITEMS` matches the gesture, then the interactor's command object is invoked instead. Thus, a gesture interactor can be configured much like a menu widget or button panel widget, by supplying a list of commands.

By default, the gesture interactor provides no visible feedback while the user is tracing out a gesture. To provide feedback, set the Am_FEEDBACK_OBJECT slot to a graphical object which has a Am_POINT_LIST slot. As the interactor records a gesture, it updates the feedback object's point list. The usual feedback object is a Am_Polygon with no fill style, which draws an unfilled, black curve as the user gestures. Here is some code that creates a feedback object:

```
Am_Polygon.Create()
     .Set (Am_FILL_STYLE, 0);
```

Note: the feedback object must be added to a window or group, or it will never be drawn.

The list below summarizes the three most common ways to use a gesture interactor, showing how the interactor's slots should be initialized in each case.

- **The gesture is uninterpreted** (used as a raw sequence of points).
  - Am_CLASSIFIER: default (0).
  - Am_ITEMS: default (0).
  - Am_COMMAND of the interactor: a command that gets the list of points from the interactor's Am_POINT_LIST slot and uses it.
  - Am_FEEDBACK_OBJECT: a Am_Polygon with no fill style which is part of a group or window.
- **The gesture is interpreted as a command**.
  - Am_CLASSIFIER: a Am_Gesture_Classifier object.
  - Am_ITEMS: a Am_Value_List of command objects whose labels correspond to the category names in the classifier.
  - Am_COMMAND of the interactor: a command that handles the case when the gesture is not recognized.
  - Am_FEEDBACK_OBJECT: a Am_Polygon with no fill style which is part of a group or window.
- **The gesture is interpreted as data**.
  - Am_CLASSIFIER: a Am_Gesture_Classifier object.
  - Am_ITEMS: default (0).
  - Am_COMMAND of the interactor: the "DO" method will access the command's Am_VALUE, which will be the name of the gesture's category, or 0 if the gesture was unrecognizable.
  - Am_FEEDBACK_OBJECT: a Am_Polygon with no fill style which is part of a group or window.

5.3.5.6.1 Creating and Using a Gesture Classifier

A gesture classifier maps gestures into named categories. Amulet uses a statistical classifier developed by Rubine[1]. In Rubine's algorithm, the classifier is *trained* by giving it a number of examples for each gesture category. Each example is converted into a feature vector (containing global features like path length and initial direction), and the examples in each category are combined to give a mean feature vector for the category and a covariance matrix for the entire classifier. After training, the classifier can be used to *recognize* a gesture. To recognize a gesture, the classifier converts it to a feature vector and determines the most likely gesture category to which it belongs (using a maximum likelihood estimator). For more information, see Rubine's thesis[1].

In Amulet, gesture classifiers are trained in a standalone application, called Agate. Agate may be found in **`samples/agate`** under the Amulet root directory. To create a classifier using Agate, add a class for each different type of gesture, giving a unique name to each class. For instance, a drawing program might have a class called "line" which contains straight-line gestures, and a class called "circle" which contains looping gestures. To demonstrate examples for a gesture class, select the class, then draw its gestures in the large empty area at the bottom of the Agate window. To produce the most forgiving classifier, try to include examples with varying size, orientation, and direction (except where your gesture classes rely on such information for uniqueness), and provide 10 to 20 examples for each class. At any point while training a classifier, you can switch from Train mode to Recognize mode in order to test the classifier. In Recognize mode, the classifier attempts to recognize the gesture you draw, highlighting the class to which it most likely belongs.

After training a classifier, save it to a file. Classifiers can be saved either with or without the examples that constructed them, depending on a checkbox in Agate's Save-File dialog. If you want to change the classifier later, then you should save it with examples, otherwise editing will be impossible. If you want to make the saved classifier as small as possible (perhaps to distribute with your application), then you can save it without examples. Either kind of file can be read into an Amulet program and used in a gesture interactor.

To use a saved classifier in an Amulet program, create an instance of the wrapper `Am_Gesture_Classifier` and load the classifier file into it. A classifier can be loaded in either of two ways: by specifying the filename in the constructor:

```
Am_Gesture_Classifier my_classifier ("my-classifier-file.cl");
```

or by opening the file as a stream and using >>:

```
Am_Gesture_Classifier my_classifier;
ifstream in("my-classifier-file.cl");
in >> the_classifier;
```

After the Am_Gesture_Classifier object is initialized with a classifier, it can be installed into the `Am_CLASSIFIER` slot of a gesture interactor:

```
gesture_interactor.Set (Am_CLASSIFIER, my_classifier);
```

---

1. Dean Rubine, *The Automatic Recognition of Gestures*, School of Computer Science, Carnegie Mellon University, December, 1991, Technical Report CMU-CS-91-202

5.3.5.6.2 Special Slots of Gesture Interactors

- Am_CLASSIFIER: a Am_Gesture_Classifier wrapper, which defines the classes of gestures that can be recognized. If this slot is 0 (the default), then gestures are left uninterpreted.
- Am_FEEDBACK_OBJECT: a graphical object that displays a trace of the gesture as it is drawn. The interactor controls the feedback object by setting its Am_POINT_LIST slot, so typically a Am_Polygon object is used. The feedback object must be part of a visible group and/or window in order to be seen.
- Am_ITEMS: a list of Am_Command objects whose Am_LABEL slots correspond to the gesture names in the classifier. When the interactor recognizes a gesture matching one of these command objects, the Am_DO_METHOD of the matching command object is invoked instead of the interactor's own command object.
- Am_POINT_LIST: set by the interactor with the list of points in the gesture. This slot is updated while the interactor is running, reflecting the points that have been traced out up to that moment. This slot can be accessed by the DO methods to get the actual points used in the gesture.
- Am_VALUE: set by the interactor with the name (a Am_String) of the recognized gesture. If the gesture is unrecognizable, this slot is set to 0.
- Am_MIN_NONAMBIGUITY_PROB: the minimum allowable probability that a recognized gesture has been correctly classified. Whenever the interactor classifies a gesture in category X, it calculates the probability that the gesture actually belongs to X. If this estimate is lower than Am_MIN_NONAMBIGUITY_PROB, then the classification is rejected, and the gesture is considered unrecognized. Intuitively, this parameter determines whether the gesture interactor will refuse to recognize gestures in the "gray areas" where gesture classes overlap, and controls how large the unrecognizable areas should be. The default for this slot is 0, so even the most ambiguous gesture is given a classification. If a stricter classifier is desired, values between 0.90 and 0.99 are reasonable. See Rubine's thesis for a full explanation.
- Am_MAX_DIST_TO_MEAN: the maximum allowable Mahalanobis distance that a recognized gesture may lie from the mean feature vector of its class. Intuitively, this parameter determines whether the gesture interactor will reject gestures that are dissimilar from all gesture classes, and controls the degree of dissimilarity required for rejection. Larger values of this slot accept more outlying gestures, but the special value of 0 (the default) turns off distance detection, so even the most outlying gesture is given a classification. If a stricter classifier is desired, values between 50 and 200 are reasonable.
- Feature slots: as part of classifying a gesture, the interactor must compute a number of features of the gesture. The results are stored by the interactor in slots of the interactor in case the command object needs them. For instance, a "selection" gesture might look at the gesture's bounding box to determine the range of the selection.
  - Am_MIN_X, Am_MIN_Y, Am_MAX_X, Am_MAX_Y: the bounding box of the path.
  - Am_TOTAL_LENGTH: the total length of the path, computed as the sum of the lengths of its segments.
  - Am_TOTAL_ANGLE: the total angle traversed by the path, in radians, computed as the sum of the signed angles of its segments (treated as vectors). For example, a zig-zag path which turns left 90 degrees, then turns right 90 degrees, has zero total

angle.

- Am_ABS_ANGLE: the total absolute angle of the path, in radians, computed as the sum of the abolute values of the segment angles.
- Am_SHARPNESS: the sum of the squares of the angles traversed, in radians. (Thus, a path which smoothly turns has a lower sharpness value than a path which makes an abrupt turn.)
- Am_START_X, Am_START_Y: the starting point on the path.
- Am_INITIAL_SIN, Am_INITIAL_COS: the sine and cosine of the initial segment of the path, i.e., the vector from the first point to the second point.
- Am_DX2, Am_DY2: the horizontal and vertical differences between the last two points.
- Am_MAGSQ2: the length of the last segment on the path.
- Am_END_X, Am_END_Y: the last point.

5.3.5.6.3 Standard operation of the Am_Gesture_Interactor

As the gesture interactor is operating, it calls various internal methods. The default operation of these methods is as follows. If this is not sufficient for your needs, then you may need to override the methods, as explained in Section 5.5.2.

While the Interactor is running, it appends the points visited by the user's mouse to the Am_Point_List in its Am_POINT_LIST slot. If a feedback object has been provided, it also appends the points to the feedback object's Am_POINT_LIST slot. If the user hits the abort key while the Interactor is running ("control_g" by default), the feedback object is made invisible and the Interactor aborts.

Otherwise, when the user performs the Am_STOP_WHEN event (usually by releasing the mouse button), then the interactor attempts to recognize the gesture using the classifier in its Am_CLASSIFER slot. If the gesture is successfully recognized, then its name is stored in Am_VALUE. Otherwise, if Am_CLASSIFIER is 0, or if the gesture cannot be recognized because it is too ambiguous (by Am_MIN_NONAMBIGUITY_PROB) or too different from the known gestures (by Am_MAX_DIST_TO_MEAN), then 0 is stored in Am_VALUE.

After classifying the gesture, the interactor looks for a command object to invoke. First, it searches the list in its Am_ITEMS slot for a command object whose Am_LABEL matches Am_VALUE (which, recall, is either the gesture name or 0 if the gesture was unrecognized). If a matching command is found, the interactor invokes its DO method; otherwise, it invokes the DO method of the command in the interactor's Am_COMMAND slot.

5.4 Advanced Features

5.4.1 Output Slots of Interactors

As they are operating, Interactors set a number of slots in themselves which you can access from the Command's DO procedure, or from constraints that determine slots of the object. The slots set by all Interactors are:

- Am_START_OBJECT: Set with the object returned by the Am_START_WHERE_TEST each time the Interactor starts. This might be useful, for example, if there are two types of "handles" connected to objects that are to be modified: one for moving and one for growing, distinguished by the value of the IS_A_MOVING_HANDLE slot. Then you might have a formula in the Am_Growing slot of a Am_Move_Grow_Interactor as follows:

```
Am_Define_Formula (bool, grow_or_move) {
   Am_Object start_object;
   start_object = self.GV(Am_START_OBJECT);
   if ((bool)start_object.GV(IS_A_MOVING_HANDLE)) return false;
   else return true;
}
```

- Am_START_CHAR: The initial Am_Input_Char that started the Interactor. This is most useful when the Am_START_WHEN slot is something like Am_ANY_KEYBOARD, or Am_ANY_MOUSE_DOWN. For example, you might put a constraint in the Am_As_Line slot that depends on which mouse button starts the Interactor. In the following, a line is created when the SHIFT key is held down, otherwise to a rectangle is created:

```
Am_Define_Formula (bool, as_line_if_shift)  {
  Am_Input_Char start_char =
     Am_Input_Char::Narrow(self.GV(Am_START_CHAR));
  if (start_char.shift) return true;
  else return false;
}
```

- Am_FIRST_X, Am_FIRST_Y: The initial X and Y locations of the mouse when the Interactor started. These are in the coordinate system of the window the initial event was in.
- Am_WINDOW: The window the Interactor is currently running in. Like graphical objects, this slot shows which window the Interactor is currently attached to. For Interactors that run over multiple windows (see Section 5.4.3), this slot is continuously updated with the current window.
- Am_CURRENT_OBJECT: The current object the Interactor is working on. This will be object returned by Am_START_WHERE_TEST when the Interactor starts running, and then by the Am_RUNNING_WHERE_TEST while the Interactor is running. This is most useful for Am_Choice_Interactors where the object the Interactor is running over changes as the Interactor runs. Most interactors use Am_RUNNING_WHERE_TEST of true, in which case the Am_CURRENT_OBJECT is set to the window.

The specific Interactors also set special slots in themselves as they are running, and then in their Command objects when they finish. These are described below in Section 5.5.3 about each specific type of Interactor.

5.4.2 Priority Levels

When an input event occurs in a window, Amulet tests the Interactors attached to objects in that window in a particular order. Normally, the correct Interactor is executed. However, there are cases where the programmer needs more control over which Interactors are run, and this section discusses the two slots which control this:

`Am_PRIORITY`: The priority of this Interactor. The default is `1.0`.

`Am_RUN_ALSO`: If `true`, then let other Interactors run after this one is completed. The default is `false`.

All the Interactors that can operate on a window are kept in a sorted list. The list is sorted first by the Interactor's priority number, and then by the display order of the graphical object the Interactor is attached to. The result is that for Interactors of the same priority, the one attached to the least covered (front-most) graphical object is handled first.

The priority of the Interactor is stored in the `Am_PRIORITY` slot and can be any positive or negative number. When an Interactor starts running, the priority level is increased by a fixed amount (defined by `Am_INTER_PRIORITY_DIFF` which is `100.0` and is defined in `inter_advanced.h`). This makes sure that Interactors that are running take priority over those that are just waiting. If you want to make sure that your Interactor runs before other default Interactors which may be running, then use a priority higher than `101.0`. For example, the debugging Interactor which pops up the inspector (see Section 5.6) uses a priority of `300.0`.

For Interactors with the same priority, the Interactor attached to the front and leaf most graphical object will take precedence. This is implemented using the slots `Am_OWNER_DEPTH` and `Am_RANK` of the graphical objects which are maintained by Opal. What this means is that an Interactor attached to a part has priority over an Interactor attached to the group that the part is in, if they both have the same value in the `Am_PRIORITY` slot. Note that this determinations does *not* take into account which objects the Interactor actually affects, just what object the Interactor is a part of. Thus, if Interactor A is attached to group G and has a `Am_START_WHERE_TEST` of `Am_Inter_In_Part`, and Interactor B is attached to part P which is in G and has a `Am_START_WHERE_TEST` of `Am_Inter_In`, then the Interactor on B will take precedence by default, even though both A and B can affect P.

If the Interactor that accepts the event has its `Am_RUN_ALSO` slot set to `true` (the default is `false`), then other Interactors will also get to process the current event. Thus, the run-also Interactor operates without grabbing the input event. This might be useful for Interactors that just want to monitor the activity in a window, say to provide a "tele-pointer" in a multi-user application. In this case, you would want the Interactor with `Am_RUN_ALSO` set to `true` to have a high priority.

Furthermore, if an Interactor that has `Am_RUN_ALSO` slot set to `false` accepts the current event, the system will continue to search to find if there are any other Interactors with `Am_RUN_ALSO` slot set to `true` that have the *same* priority as the Interactor that is running. These are also allowed to process the current event.

5.4.3 Multiple Windows

A single Interactor can handle objects which are in multiple windows. Since an Interactor must be attached to a single window, graphical object or group, a special mechanism is needed to have an Interactor operate across multiple windows. This is achieved by using the Am_MULTI_OWNERS slot. The value of this slot can be:

- false or zero: which means that this slot is ignored, and the Interactor only works on the window of the object it is attached to. This is the default.
- true (or any non-zero integer): the Interactor operates on *all* windows created using Amulet, now or in the future.
- a Am_Value_List containing a list of objects or windows, which means that the Interactor works in all the windows of the objects. This list *must include* the "main" window of the object that the interactor is part of.

Of course, the function in the Am_START_WHERE_TEST slot must search for objects in all of the appropriate windows. It might use the value of the Am_WINDOW slot of the Interactor, which will contain the window of the current event. This is normally sufficient for Am_Choice_Interactors, Am_One_Shot_Interactors, and Am_Text_Edit_Interactors that want to operate on objects in multiple windows. For these interactors, the Am_MULTI_OWNERS slot can be either a list of windows or a list of objects in those windows. Be sure to include the window to which the interactor is attached.

Special features are built-in to support interactors that might want to *move* an object from one window to another, such as Am_Move_Grow_Interactors, Am_New_Points_Interactors, and Am_Gesture_Interactors. In this case, the Am_MULTI_OWNERS slot should contain a list of objects which should serve as the owners of the graphical objects as they are moved from one window to another. Then, the interactor will automatically search the Am_MULTI_OWNERS list for an object in the window that the cursor is currently in, and if found, then the object being moved (returned by the Am_START_WHERE_TEST) will change to have that object as its owner.

Often the feedback objects should be in a different owner than the "real" objects being moved. In this case, you can set the Am_MULTI_FEEDBACK_OWNERS slot with a list of owner objects for the feedback object. In this case, the feedback object of the interactor (specified in the Am_FEEDBACK_OBJECT slot of the interactor) will automatically be changed to be in the object of the appropriate window. If the Am_MULTI_FEEDBACK_OWNERS slot is NULL, then the owners in the Am_MULTI_OWNERS slot are used for the feedback object as well.

The Am_FEEDBACK_OBJECT can also be a top-level window object, in which case the various types of interactor objects will move the object around on the screen. This is particularly useful when you want to be sure to see the feedback even when the cursor is not over an Amulet window. For move-grow interactors when using a window as the feedback, you should use something like Am_ATTACH_NW for the Am_WHERE_ATTACH slot.

Examples of using multi-window interactors are in the test file inter/testinter.cc.

5.4.4 Running_Where

Section 5.3.2 mentioned that Interactors can be defined so that they stop operating when the mouse goes outside of their active area. The active area is defined by the value of the Am_RUNNING_WHERE_OBJECT slot. This slot should contain either a graphical object or true, which means anywhere (so the Interactor never goes outside). The default for most Interactors for this slot is true, but for Choice Interactors, this slot contains a constraint that makes it have the same object as where the Interactor starts. To refine where the Interactor should be considered outside, the programmer can also supply a value for the Am_RUNNING_WHERE_TEST slot, which defaults to Am_Inter_In except for choice Interactors, where it contains a constraint that uses the same function as the Am_START_WHERE_TEST. We have found that programmers rarely need to specify the running where of Interactors.

5.4.5 Starting, Stopping and Aborting Interactors

Interactors normally start, stop and abort due to actions by the user, but it is sometimes useful for the programmer to be able to explicitly control the Interactors. The following functions are useful for controlling this. If you have a widget, you would use the corresponding widget functions instead (Am_Start_Widget, Am_Stop_Widget, and Am_Abort_Widget -- Section 5.5).

```
extern void Am_Abort_Interactor(Am_Object inter);
```

Am_Abort_Interactor causes the Interactor to abort (stop running). The command associated with the Interactor is not queued for Undo.

```
extern void Am_Stop_Interactor(Am_Object inter,
                       Am_Object stop_obj = Am_No_Object,
                       Am_Input_Char stop_char = Am_Default_Stop_Char,
                       Am_Object stop_window = Am_No_Object, int stop_x = 0,
                       int stop_y = 0);
```

Am_Stop_Interactor explicitly stops an interactor as if it had completed normally (as if the stop event had happened). The command associated with the interactor is queued for Undo, if appropriate. If the interactor was not running, Am_Stop_Interactor raises an error. If the stop_obj parameter is not supplied, then Am_Stop_Interactor uses the last object the interactor was operating on (if any). stop_char is the character sent to the Interactor's routines to stand in for the final event of the Interactor. If stop_window is not supplied, then will use stop_obj's window, and stop_x and stop_y will be stop_obj's origin. If stop_window is supplied, then it should be the window that the final event is with respect to, and you must also supply stop_x and stop_y as the coordinates in the window at which the interactor should stop.

```
extern void Am_Start_Interactor(Am_Object inter,
                Am_Object start_obj = Am_No_Object,
               Am_Input_Char start_char = Am_Default_Start_Char,
                Am_Object start_window = Am_No_Object, int start_x = 0,
```

```
                    int start_y = 0);
```

Am_Start_Interactor is used to explicitly start an Interactor. If the interactor is already running, this does nothing. If start_obj is not supplied, then will use the inter's owner. If start_window is not supplied, then uses start_obj's and sets start_x and start_y to start_obj's origin. start_char is the initial character to start the interactor with. If start_window is supplied, then you must also supply start_x and start_y as the initial coordinate (with respect to the window) for the Interactor to start at.

If you want an interactor to never start by itself, you can have the Am_START_WHEN slot be NULL, and then start it explicitly using the Am_Start_Interactor procedure. Similarly, the Am_STOP_WHEN slot can be NULL for an Interactor that should never stop by itself.

5.4.6 Support for Popping-up Windows and Modal Windows

It is often useful to be able to pop up a window, and wait for the user to respond. This can eliminate having to chain a number of DO methods together when you just want to ask the user a question. Amulet provides low-level functions to support this. If the window to ask the question is one of a few standard types, you might alternatively use the built-in dialog boxes described in Section 6.3 of the widgets chapter.

The way the low level routines are used is that the programmer calls Am_Pop_Up_Window_And_Wait passing in the window to wait on, and then that window contains a number of widgets, at least one of which (usually the "OK" button) calls Am_Finish_Pop_Up_Waiting as part of its DO method. The value passed to Am_Finish_Pop_Up_Waiting is then returned by the Am_Pop_Up_Window_And_Wait call.

```
extern void Am_Pop_Up_Window_And_Wait(Am_Object window,
                                      Am_Value &return_value,
                                      bool modal = true);
```

Am_Pop_Up_Window_And_Wait sets the visible of the window to true, and then waits for a routine in that window to call Am_Finish_Pop_Up_Waiting on that window. Returns the value passed to Am_Finish_Pop_Up_Waiting by setting the return_value parameter. If modal is true, the default, then the user will only be able to work on this one window, and all other windows of this application will be frozen (input attempts to other windows will beep). Note that input can still be directed to other applications, unlike modal dialog boxes on the Macintosh.

It is legal to call Am_Pop_Up_Window_And_Wait from inside a DO method, so a modal window might pop up another modal window.

```
extern void Am_Finish_Pop_Up_Waiting(Am_Object window,
                                     Am_Value return_value);
```

`Am_Finish_Pop_Up_Waiting` sets `window`'s `Am_VISIBLE` to FALSE, and makes the `Am_Pop_Up_Window_And_Wait` called on the same window return with the value passed as the `return_value`. Although there is no default, it is acceptable to pass `Am_No_Value` as the return value.

This allows code like:

```
Am_Value val;
Am_Pop_Up_Window_And_Wait(my_query_window, val);
if (val.Valid()) ...
else ...;
```

5.5 Customizing Interactor Objects

Amulet allows the programmer to customize the behavior of Interactors at multiple levels. As described in the previous sections, many aspects of the behavior of Interactors can be controlled through the parameters of the Interactors. We believe that in almost all cases, programmers will be able to create their applications by using these built-in parameters of the pre-defined types of Interactors. If you are happy with the standard behavior, but want some *additional* actions to happen when the interactor is finished, then you can attach a custom Command object to the interactor, as described in Section 5.5.1. If you want behavior similar to a standard Interactor,. but slightly different, then you might want to override some of the standard methods that implement the Interactor's behavior, as described in Section 5.5.2. However, there might be rare cases when an entirely new type of Interactor is required, as described in Section 5.5.3. For example, in Garnet which had a similar Interactor model, *none* of the applications created using Garnet needed to create their own Interactor types. However, when the Garnet group wanted to add support for Gesture recognition, this required writing a new Interactor. Since Amulet is designed to support investigation into new interactive styles and techniques, new kinds of Interactors may be needed to explore new types of interaction beyond the conventional direct manipulation styles supported by the built-in Interactors. In summary, we feel you should only need to create a new kind of Interactor when you are supporting a radically different interaction style.

5.5.1 Adding Behaviors to Interactors

If you want the standard behavior of an interactor *plus* some additional behavior, you can override the methods of the Command object in the Interactor. (See Section 5.6 for a complete discussion of Command objects.) The command object in the Interactors all have empty methods, so you can override the methods without concern. The methods that you can override include:

- `Am_START_DO_METHOD`: Called once each time the Interactor starts running.
- `Am_INTERIM_DO_METHOD`: Called each time there is an input event.
- `Am_ABORT_DO_METHOD`: Called when the Interactor should abort.
- `Am_DO_METHOD`: Called when the stop action happens and the Interactor should terminate normally.

There are three different kinds of methods that you can put into any of these slots of the command objects:

- Am_Object_Method: which just takes the command object as a parameter.
- Am_Mouse_Event_Method: which takes the command object, the mouse x and y and ref_obj of the command, and the input character. For example:

```
Am_Define_Method(Am_Mouse_Event_Method, void, my_mouse_method,
                 (Am_Object inter_or_cmd, int mouse_x, int mouse_y,
                  Am_Object ref_obj, Am_Input_Char ic));
```

- Am_Current_Location_Method: which takes the command object, the object_modified, which is usually the object that the Interactor is modifying, and the description of the current input event position (see Section 5.3.3.4). **This type of method can only be used for move_grow and new_point interactors.** An example:

```
Am_Define_Method(Am_Current_Location_Method, void, my_do_method,
      (Am_Object inter, Am_Object object_modified,
       Am_Inter_Location data)) { ... }
```

Note that the Am_Mouse_Event_Method ***and the*** Am_Current_Location_Method ***cannot be used for command objects in widgets, only for the command objects in Interactors. The command objects in widgets only support*** Am_Object_Method.

In addition to the information passed as parameters to the command object, slots of the command object and of the Interactor are set by the Interactor and may be of use to the methods. For all command objects, the following slots are available:

In the command objects:

- Am_OWNER: While the methods are running, the Am_OWNER of the command is the Interactor, so you can use Get_Owner() to access the slots. However, the command object is no longer part of the Interactor at Undo time, so all of the UNDO methods should use the Am_SAVED_OLD_OWNER slot instead.
- Am_SAVED_OLD_OWNER: Set with the Interactor. This stays the Interactor the command was attached to, so you can use this in Undo methods.

Each of the types of Interactor sets specific slots in the Interactor that the command object might want to access, as described in the following sections. **Remember that these slots are set in the Interactor, not in the Command object.** The slots described below are in addition to the slots described with the specific interactors in Section 5.3.5 and the slots set into all interactors, described in Section 5.4.1.

5.5.1.1 Available slots of Am_Choice_Interactor and Am_One_Shot_Interactor

- Am_OLD_INTERIM_VALUE which is set with an object or NULL which is the previously interim selected object.
- Am_INTERIM_VALUE which is set with the newly selected object.
- Am_VALUE which is set with the final result which may be NULL, an object (if only a single object can be selected) or a Am_Value_List of objects.

- Am_OLD_VALUE which is set with the previous value of Am_VALUE for use if the command is undone.

5.5.1.2 Available slots of Am_Move_Grow_Interactors

- Am_OBJECT_MODIFIED which is the object being moved or changed size.
- Am_INTERIM_VALUE which contains a Am_Inter_Location of the current position and size of the object.
- Am_OLD_VALUE which holds a copy of the old (original) value as an Am_Inter_Location, used in case the Interactor is aborted or later undone.
- Am_VALUE (only available to the DO and various UNDO methods) which contains the final Am_Inter_Location used to set the position and size of the object.

5.5.1.3 Available slots of Am_New_Point_Interactors

- Am_TOO_SMALL (a bool) which is set by the interactor if the size is currently smaller than the minimum allowed. The default method turns off the feedback if Am_TOO_SMALL is true.
- Am_INTERIM_VALUE which contains a Am_Inter_Location for the current position and size of the feedback object.
- Am_VALUE (only available in the DO and various UNDO methods) which holds the object which was created as a result of this Command.

5.5.1.4 Available slots of Am_Text_Interactors

- Am_OBJECT_MODIFIED - the object being edited, which will be an instance of Am_Text.
- Am_OLD_VALUE - the original string for the object, as a Am_String, in case the user aborts or calls Undo.
- Am_VALUE (only available to the DO and various UNDO methods) which is the new (final) string for the object.

5.5.1.5 Available slots of Am_Gesture_Interactors

- Am_POINT_LIST - the list of points so far.

5.5.2 Modifying the Behavior of the Built-in Interactors

If you need to override the standard behavior of an Interactor, and the supplied parameterization is not sufficient, then you may need to override one of the standard methods of the Interactor that control its behavior. All Interactors internally use the same state machine, as shown in the Figure in Section 5.3.2. At each input event, Amulet will call an internal method of the Interactor, which typically sets some internal slots of the Interactor, and first calls a customizable method described below, and then calls the corresponding method of the command object, described in Section 5.5.1. The methods of the Interactor you can override are the same as the methods in the command object described above:

- Am_START_DO_METHOD: Called once each time the Interactor starts running.
- Am_INTERIM_DO_METHOD: Called each time there is an input event.
- Am_ABORT_DO_METHOD: Called when the Interactor should abort.
- Am_DO_METHOD: Called when the stop action happens and the Interactor should terminate normally.

As with the command objects in Interactors, the methods directly in the Interactors can take one of the three types: Am_Object_Method, Am_Mouse_Event_Method, or Am_Current_Location_Method, as defined in Section 5.5.1.

Remember that if you override the various DO methods, this will remove the standard behavior of the Interactors, so there may be some of the behavior you may want to re-implement in your code. For example, the Am_ABORT_DO_METHOD and the Am_DO_METHOD take care of making the feedback object (if any) invisible. The Am_ABORT_DO_METHOD is also responsible for restoring the object to its original state.

5.5.3 Entirely New Interactors

This section gives an overview of the how to build an entirely new type of Interactor. As said above, we believe this almost never be necessary. You may need to look at the source code for one of the built-in Interactors to see how they operate in detail.

The main event loop in Amulet takes each input event and looks at the sorted list of Interactors with each window, and then asks each Interactor in turn if they want to handle the input event. This is done by sending the Interactor one of the messages listed below, based on the current state of the Interactor, held in the slot Am_CURRENT_STATE (see the state machine figure in Section 5.3.2). The different Interactors are distinguished by having different functions for these messages. All of the messages are of type Am_Inter_Internal_Method (defined in inter_advanced.h). Most of these methods set up some slots of the Interactor, and then call the appropriate Interactor DO method. The specific internal methods you need to write for a new type of Interactor are stored in the following slots:

- Am_INTER_START_METHOD: This is called when the Interactor should first start running. Typically, the method would initialize various fields and then call the Am_START_DO_METHOD of the Interactor and then the Command object. It will then call the Am_INTERIM_DO_METHOD of the Interactor and then the Command object on the first point.
- Am_INTER_RUNNING_METHOD: This is called for each incremental mouse movement or keyboard key while the Interactor is executing. Typically, it would set some slots and then call the Am_INTERIM_DO_METHOD of the Interactor and then the Command object.
- Am_INTER_OUTSIDE_METHOD: This is called if the mouse moves outside of the Am_RUNNING_WHERE_OBJECT while the Interactor is running. Typically it will call the Am_ABORT_DO_METHOD of the Interactor and then the Command object.

- Am_INTER_BACK_INSIDE_METHOD: This is called if the mouse moves back inside the Am_RUNNING_WHERE_OBJECT while the Interactor is running. Typically it will call the Am_START_DO_METHOD followed by the Am_INTERIM_DO_METHOD of the Interactor and then the Command object.
- Am_INTER_ABORT_METHOD: This is called when the user executes the abort key to cause the Interactor to abort while it is executing. Typically, this will call the Am_ABORT_DO_METHOD of the Interactor and then the Command object.
- Am_INTER_STOP_METHOD: This is called when the Interactor stops (finishes). Typically it will set some slots in the Command object and then call the Am_DO_METHOD of the Interactor and then the Command object.
- Am_INTER_OUTSIDE_STOP_METHOD: This is called when the user executes the stop event while the Interactor is outside. For all the built-in Interactors, this method is not needed because the default is to call the Am_INTER_ABORT_METHOD.

5.6 Command Objects

Unlike other toolkits where the widgets call "call-back" procedures, the widgets and Interactors in Amulet allocate *Command Objects* and call their "Do" methods. Whereas so far this is pretty much equivalent, Command Objects also have slots that handle undoing, enabling and disabling, and help. Command objects must be added as *parts* of the objects they are attached to, so every Interactor and widget has a part named Am_COMMAND which contains a Am_Command object. For all widgets and Interactor commands, the default methods are empty, so you can freely supply any method you want. There are also a set of command objects supplied in the library which you might be able to use in applications, as described in Section 6.4 of the Widgets chapter. If you create your own custom command objects, be sure to create UNDO methods, as described in Section 5.6.2.

The top-level definition of a Command object is:

```
Am_Command = Am_Root_Object.Create ("Am_Command")
  .Set (Am_DO_METHOD, NULL)
  .Set (Am_UNDO_METHOD, NULL)
  .Set (Am_REDO_METHOD, NULL)
  .Set (Am_SELECTIVE_UNDO_METHOD, NULL)
  .Set (Am_SELECTIVE_REPEAT_SAME_METHOD, NULL)
  .Set (Am_SELECTIVE_REPEAT_ON_NEW_METHOD, NULL)
  .Set (Am_SELECTIVE_UNDO_ALLOWED, Am_Standard_Selective_Allowed)
  .Set (Am_SELECTIVE_REPEAT_SAME_ALLOWED, Am_Standard_Selective_Allowed)
  .Set (Am_SELECTIVE_REPEAT_NEW_ALLOWED,
  Am_Standard_Selective_New_Allowed)

  .Set (Am_ACTIVE, true)
  .Set (Am_LABEL, "A command")
  .Set (Am_SHORT_LABEL, 0) //if 0 then uses Am_LABEL
 .Set (Am_ACCELERATOR, 0) // event to also execute this
 .Set (Am_ID, 0) // if non-zero, identifies the cmd instead of label
  .Set (Am_VALUE, 0)
  .Set (Am_OLD_VALUE, 0) //usually for undo
  .Set (Am_OBJECT_MODIFIED, 0)
  .Set (Am_SAVED_OLD_OWNER, NULL)
  .Set (Am_IMPLEMENTATION_PARENT, 0)
  ;
```

Most Command objects supply a Am_DO_METHOD procedure which is used to actually execute the Command. It will typically also store some information in the Command object itself (often in the Am_VALUE and Am_OLD_VALUE slots) to be used in case the Command is undone. The Am_UNDO_METHOD procedure is called if the user wants to undo this Command, and usually swaps the object's current values with the stored old values. The Am_REDO_METHOD procedure is used when the user wants to undo the undo. Often, it is the same procedure as the Am_UNDO_METHOD. The various SELECTIVE_ methods support selective undo and repeat the command, as explained in Section 5.6.2.3. The Am_ACTIVE slot controls whether the Interactor or widget that owns this Command object should be active or not. This works because widgets and Interactors have a constraint in their active field that looks at the value of the Am_ACTIVE slot of their Command object. Often, the Am_ACTIVE will contain a constraint that depends on some state of the application, such as whether there is an object selected or not. The Am_LABEL slot is used for Command objects which are placed into buttons and menus to show what label should be shown for this Command. If supplied, the Am_SHORT_LABEL is used in the Undo dialog box to label the command. For commands in button and menu widgets, the Am_ACCELERATOR slot can contain an Am_Input_Char which will be used as the accelerator to perform the command (see Section 6.2.3.2 of the Widgets chapter). For commands in button widgets, the widgets use the command to determine the value to return when the button is hit. If the Am_ID slot is set, then this value, which can be of any type, is used. If Am_ID is 0, then the value of the Am_LABEL slot is returned.

Various slots are typically set by the DO method for use by the UNDO methods. The Am_VALUE slot is set with the value for use. You have to look at the documentation for each Interactor or Widget to see what form the data in the Am_VALUE slot is. The DO method typically also sets the Am_OBJECT_MODIFIED slot with the object (or a Am_Value_List of objects) that the Command affects. The Am_SAVED_OLD_OWNER slot is set by the Interactors and Widgets to contain the Interactor and widget itself. Finally, the Am_IMPLEMENTATION_PARENT supports the hierarchical decomposition of commands, as described in the next section (Section 5.6.1).

As mentioned above in Section 5.5.1, command objects which are used in Interactors also support the Am_START_DO_METHOD, Am_INTERIM_DO_METHOD, and the Am_ABORT_DO_METHOD.

5.6.1 Implementation_Parent hierarchy

Normal objects are part of two hierarchies: the prototype-instance hierarchy and the part-owner hierarchy. The Command objects has an *additional* hierarchy defined by the Am_IMPLEMENTATION_PARENT slot. Based on the Ph.D. research of David Kosbie[1], we allow lower-level Command objects to invoke higher-level Command objects. For example, the Command object attached to a move-grow Interactor which is allowing the user to move a scroll bar indicator calls the Command object attached to the scrollbar itself.

1. David S. Kosbie and Brad A. Myers, "Extending Programming By Demonstration With Hierarchical Event Histories" *The 1994 East-West International Conference on Human-Computer Interaction*. St. Petersburg, Russia, August, 1994. pp. 147-157.

This novel model for Command objects is more completely described in a conference paper[1] which is available from the Amulet WWW site (`http://www.cs.cmu.edu/~amulet/papers/commandsCHI.html`).

Simply, the system calls the `Am_DO_METHOD` of the lowest level command associated with the Interactor or widget, and then looks in the `Am_IMPLEMENTATION_PARENT` slot of that command. If that slot contains a command object, then the `Am_DO_METHOD` of that command is also called, and so on. Internally, all the widgets use this mechanism to chain commands together, and you can use it also to execute multiple commands on an event.

When a command is queued for undo, *all* the commands along the `Am_IMPLEMENTATION_PARENT` chain are queued. When the user requests an undo or selective undo or redo, then all the commands in the `Am_IMPLEMENTATION_PARENT` chain are undone in the *same* order as they were originally executed (from child to parent). This is in case lower-level commands set state which is used by higher-level commands.

The advantage of this design is that the low-level Command objects do not need to know how they are being used, and can just operate normally, and the higher-level Command objects will update whatever is necessary. Note that the `Am_IMPLEMENTATION_PARENT` hierarchy is not usually the same as the part-owner hierarchy. Unfortunately, it seems to be difficult or impossible for Amulet to deduce the parent hierarchy from the part-owner hierarchy, which is why the programmer must explicitly set the `Am_IMPLEMENTATION_PARENT` slot when appropriate. Of course, the built-in widgets (like the scroll bar) have the internal Command objects set up appropriately.

You might use the `Am_IMPLEMENTATION_PARENT` slot for the Command object in the 'OK' button widget inside a dialog box, so the OK widgets's action will automatically call the dialog box's Command object.[2] Another example is that for the button panel widget (see the Widgets chapter), you can have a Command object for each individual item and/or a Command object for the entire panel. If you want the individual item's Command to be called *and* the top-level Command to be called, then you would make the top-level Command be the `Am_IMPLEMENTATION_PARENT` of each of the individual item Commands.

5.6.2 Undo

All of the Command objects built into the Interactors and widgets automatically support full undo, redo and selective undo and repeat. This means that the default `Am_DO_METHOD` procedures store the appropriate information. Built-in "undo-handlers" know how to copy the command objects when they are executed, save them in a list of undoable actions, and execute the undo, redo and selective methods of the commands. Thus, to have an application support undo is generally a simple process. You need to create an undo-handler object and attach it to a window, and then have some button or menu item in your application call the undo-handler's method for undo, redo, etc.

---

1. Brad A. Myers and David S. Kosbie. "Reusable Hierarchical Command Objects," *Proceedings CHI'96: Human Factors in Computing Systems.* Vancouver, BC, Canada. April 14-18, 1996. To Appear.
2. The conference paper on commands also discusses another chain using the `Am_COMPOSITE_PARENT` slot which is useful for dialog boxes. This chain is not implemented in the released version of Amulet.

5.6.2.1 Enabling and Disabling Undoing of Individual Commands

If there are operations in the application that are *not* undoable, for example like File Save, then you should have the Am_UNDO_METHOD and Am_REDO_METHOD slots as null. As explained below, there are special methods that determine whether the command is selective undoable and repeatable.

If there are operations that should not go on the undo list at all, for example like scrolling, there is an easy way to specify this. Simply set the Am_IMPLEMENTATION_PARENT slot of the top-level command to be the constant value Am_NOT_USUALLY_UNDONE, which is defined in inter.h. (All commands whose Am_IMPLEMENTATION_PARENT slot is null are assumed to be top-level commands and are queued for undo.)

5.6.2.2 Using the standard Undo Mechanisms

There are three styles of undo supplied by Amulet. These are described in this and the next sections. Section 5.6.2.3 discusses how programmers can implement other undo mechanisms. The undo mechanisms are implemented using Am_Undo_Handler objects.

The three kinds of undo supplied by Amulet are:

- **Single undo**, like on the Macintosh. This is implemented using the Am_Single_Undo_Object. This handler supports undoing a single command. The last operation can be undone, and the last undone operation can be redone. As soon as another operation is performed, the previous Command is discarded so it can no longer be undone or redone.
- **Multiple undo**, like in Microsoft Word V6. This is implemented using the Am_Multiple_Undo_Object. This handler supports undoing an arbitrary number of Commands, all the way back to the first command. This is implemented by saving all the commands executed since the application is started, so the list can grow quite long. If a command is undone, then it can be redone, but only the last undone command is saved. Thus, after undoing a series of commands, after undoing the last undo (redo), there is nothing available to redo. The Undo itself is not part of the command history.
- A novel form of **Selective undo**, where, in addition to the multiple undo, it also allows previous commands to be selected and explicitly undone or repeated, as described in Section 5.6.2.3. This is also implemented by the Am_Multiple_Undo_Object.

To make an application support undo, it is only necessary to put an instance of an undo handler object into the Am_UNDO_HANDLER slot of a window. For example:

```
my_win = Am_Window.Create("my_win")
  .Set (Am_LEFT, ...)
  ...
  .Set (Am_UNDO_HANDLER, Am_Multiple_Undo_Object.Create("undo"))
```

You can put the same undo-handler object into the Am_UNDO_HANDLER slot of multiple windows, if you want a single list of undo actions for multiple windows (for example, for applications which use multiple windows). Now, all the Commands executed by any widgets or Interactors that are part of this window will be automatically registered for undoing.

Next, you need to have a widget that will allow the user to execute the undo and redo. The Am_Undo_Command object provided by widgets.h encapsulates all you typically need to support undo, and the Am_Redo_Command supports redo (see Section 5.6 of the widgets chapter). There is also a dialog box provided to support selective undo (next section).

If the built-in undo and redo commands are not sufficient, then you can easily create your own. The command's Am_ACTIVE slot should depend on whether the undo and redo are currently allowed. The undo_handler objects provide the Am_UNDO_ALLOWED slot to tell whether undo is allowed. This slot contains the command object that will be undone (in case you want to have the label of the Undo Command show what will be undone). The Am_REDO_ALLOWED slot of the undo object tells whether redo is allowed and it also will contain a command object or NULL. To actually perform the undo or redo, you call the method in the Am_PERFORM_UNDO or Am_PERFORM_REDO slots of the undo object. These methods are of type Am_Object_Method.

5.6.2.3 The Selective Undo Mechanism

In addition to the regular multiple-undo mechanism, Amulet now supports a novel selective undo mechanism. This allows the user to select previously executed commands from the history and undo or repeat them. See the conference paper mentioned above for a complete description.

We use the terminology from Windows and the Macintosh, where "undo" means to undo the command, "redo" means to undo-the-undo (make the command no longer be undone), and "repeat" to do the command again. The mechanism supports the conventional undo and redo, as described above. In addition, the user can select a command from a visible list. At this point, the user can ask that the command be undone, it can be repeated on the original objects, or repeated on the currently selected objects. For example, for a command that changes an object that was originally blue to be red, undo would clearly restore the object to be blue, and redo would make it red again. Imagine the object was later turned green. Selective undo of the original change-color command would restore the object to be blue, and selective repeat would make it be red again. Since the selective commands add a new command to the history list, the user might undo the selective operation itself, which would make the object be green again. If a new object was selected, selectively repeating the original command on this new object would make it be red.

All of the built-in commands support the complete selective undo mechanism. The next section describes what you need to implement to create your own commands that support selective undo. Next is described the commands you would add to your application to allow access to the selective undo features.

5.6.2.3.1 Supporting Selective Undo in your own Command Objects

Regular undo and redo operate on the original command object, and remove or add it to the undo list. However, selective undo and repeat create a *copy* of the original command object, and then add this copy to the top of the undo list. Thus, using the regular undo method or selective undo of the most recent command should both have the same effect to the application, they have very different affects on the undo list. (Thus, you can implement an undo like in the Emacs editor, where all undo's are added to the command history, by simply always using selective undo instead of the "regular" undo.) This copying is automatically handled by the default undo handlers (including renaming the command to have "Undo" or "Repeat" in front of the name).

When a selective undo or repeat operation is requested by the end user, the standard undo handler first makes a copy of the command object, and then executes the appropriate methods. This means that the methods are free to set local data into the commands to support possible subsequent undo of the command.

The messages in each command that support selective undo are:

- `Am_SELECTIVE_UNDO_ALLOWED`: This method determines whether the command can be undone given the current context. For example, if the command was a change color, this method might check whether the object operated on is still visible. This method is of type `Am_Selective_Allowed_Method` which takes the command object and returns a boolean. Most commands will find the default method, `Am_Standard_Selective_Allowed`, is sufficient for their needs, if they store the information into the standard slots. This method checks whether the object or list of objects in the `Am_OBJECT_MODIFIED` slot of the command object is valid and still visible in the window. However, some commands, like deleting, are undoable when the object is *not* visible, so these commands will need a custom `Am_SELECTIVE_UNDO_ALLOWED` method.
- `Am_SELECTIVE_UNDO_METHOD`: This method performs the undo on the original object or objects. The method is of the type `Am_Object_Method` and just takes the command object to be undone.
- `Am_SELECTIVE_REPEAT_SAME_ALLOWED`: This determines whether the command can be repeated on the original object. The type of the method is `Am_Selective_Allowed_Method`, and the default method is `Am_Standard_Selective_Allowed`, which is the same as for the `Am_SELECTIVE_UNDO_ALLOWED` method.
- `Am_SELECTIVE_REPEAT_SAME_METHOD`: This does the command again. The method is of the type `Am_Object_Method` and just takes the command object to be undone.
- `Am_SELECTIVE_REPEAT_NEW_ALLOWED`: This returns true if the command can be selectively redone on a new object or list of objects (typically, the current selection). The type of the method is `Am_Selective_New_Allowed_Method`, which takes the command object, the new object or list of objects to be operated on, and returns a boolean. The default method is `Am_Standard_Selective_New_Allowed`, which determines whether the new object is valid and visible.

- Am_SELECTIVE_REPEAT_ON_NEW_METHOD: This performs the repeat on the new object, and is of type Am_Selective_Repeat_New_Method, which takes the command object and the new object or objects selected.

In addition to the methods described above, additional methods that might be useful that are provided include:

- Am_Standard_Selective_Return_True: can be used in the Am_SELECTIVE_UNDO_ALLOWED and Am_SELECTIVE_REPEAT_SAME_ALLOWED slots to always return true (the command is always allowed).
- Am_Selective_Allowed_Return_False: same as above, except to always return false (the command is never allowed)
- Am_Selective_New_Allowed_Return_True: can be used in the Am_SELECTIVE_REPEAT_NEW_ALLOWED to always return true.
- Am_Selective_New_Allowed_Return_False: can be used in the Am_SELECTIVE_REPEAT_NEW_ALLOWED to always return false.

5.6.2.3.2 Interface to Selective Undo in Applications: Undo Dialog Box

Amulet comes with an experimental interface to selective undo, in the form of the Am_Undo_Dialog_Box object which implements one form of undo dialog box. This is exported from undo_dialog.h (or undo_dia.h on the PC). You can see an example of this dialog box in the test file testselectionwidgets.cc (or testselw.cpp on the PC).

This dialog box is not part of the standard system, so you have to initialize it explicitly using Am_Initialize_Undo_Dialog_Box(). You can then create an instance of the Am_Undo_Dialog_Box, and set its required slots, which are:

- Am_UNDO_HANDLER_TO_DISPLAY: Set with the undo handler from the main window.
- Am_SELECTION_WIDGET: Set with the selection widget (see Section 6.2.6 of the Widgets chapter) for the main window.
- Am_SCROLLING_GROUP: If this is set with a scrolling group, then the Am_Undo_Dialog_Box will allow scrolling in that window to be queued for undoing or redoing.

To display the undo dialog box, you might use the Am_Show_Undo_Dialog_Box_Command from undo_dialog.h. This command requires that the dialog box object be put into the Am_UNDO_DIALOG_BOX slot of the command object.

For example, the set up of the undo dialog box in the test program testselectionwidgets.cc is:

```
Am_Initialize_Undo_Dialog_Box();
my_undo_dialog = Am_Undo_Dialog_Box.Create("My_Undo_Dialog")
    .Set(Am_LEFT, 550)
    .Set(Am_TOP, 200)
    .Set(Am_UNDO_HANDLER_TO_DISPLAY, undo_handler)
    .Set(Am_SELECTION_WIDGET, my_selection)
    .Set(Am_SCROLLING_GROUP, scroller)
    .Set(Am_VISIBLE, false)
    ;
```

```
Am_Screen.Add_Part(my_undo_dialog); //don't forget to add the db to the screen
menu_bar = Am_Menu_Bar.Create("menu_bar")
      .Set(Am_ITEMS, Am_Value_List ()
           ...
              .Add(Am_Show_Undo_Dialog_Box_Command.Create()
                          .Set(Am_UNDO_DIALOG_BOX, my_undo_dialog))
        ;
```

5.6.2.4 Building your own Undo Mechanisms

Usually, one of the two the supplied Undo objects will do what you want, but Amulet is designed to be easily extensible with new kinds of undo mechanisms. For example, you might want to support arbitrary numbers of redo's or put a limit on the number of Commands that can be undone. To implement these, you would make your own undo handler object as an instance of `Am_Undo_Handler` and supply values in the following slots:

- `Am_REGISTER_COMMAND`, a method to be called to register a newly executed Command. This method should correctly handle the case when the command has an Implementation Parent command object, in which case the command will usually not be queued directly since it will be queued indirectly when the top-level command in the implementation-parent chain is queued. The method in this slot is of type `Am_Register_Command_Method`.
- `Am_PERFORM_UNDO`, a method of type `Am_Object_Method` to be called to execute the undo of the next command to be undone.
- `Am_PERFORM_REDO`, a method of type `Am_Object_Method` to be called to execute the redo of the last undone Command.
- `Am_UNDO_ALLOWED`, should contain a formula which returns a command object or NULL, to say whether undo is currently allowed and if so, on which command object it will be performed.
- `Am_REDO_ALLOWED`, should contain a formula which returns a command object or NULL, to say whether redo (undo the undo) is currently allowed and if so, on which command object it will be performed.

If your undo handler will support selective undo, then the following slots should also be supported:

- `Am_SELECTIVE_UNDO_ALLOWED`, a method of type `Am_Selective_Allowed_Method`, which takes a command object and returns true if that command can be selectively undone.
- `Am_SELECTIVE_UNDO_METHOD`, a method of type `Am_Handler_Selective_Undo_Method` to be called to perform the selective undo.
- `Am_SELECTIVE_REPEAT_SAME_ALLOWED` which contains a method of type `Am_Selective_Allowed_Method` which returns true if the specified command can be selectively repeated.
- `Am_SELECTIVE_REPEAT_SAME_METHOD` which contains a method of type `Am_Handler_Selective_Undo_Method` and which performs the repeat.
- `Am_SELECTIVE_REPEAT_NEW_ALLOWED` which contains a method of type `Am_Selective_New_Allowed_Method` and returns true if the specfied command can be repeated on the specified new object or objects.

- Am_SELECTIVE_REPEAT_ON_NEW_METHOD which contains a method of type Am_Handler_Selective_Repeat_New_Method which performs the repeat on new.

5.7 Debugging

The Inspector and Interactors provide a number of mechanisms to help programmers debug programs. The primary one is a *tracing* mechanism that supports printing to standard output (cout) whenever an "interesting" Interactor or Command event happens. Amulet supplies many options for controlling when printout occurs, as described below. You typically would set these in the Inspector, but you can also set these parameters in your code.

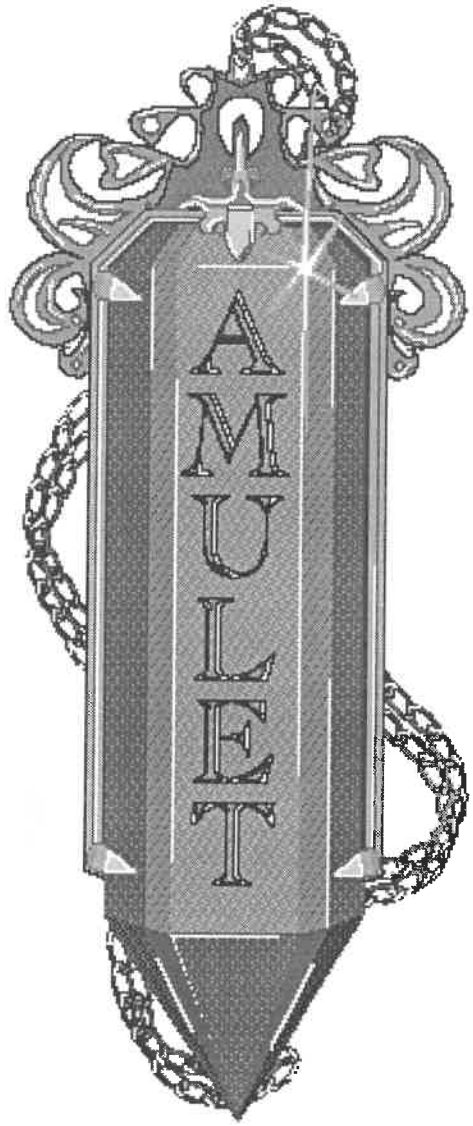
```
typedef enum { Am_INTER_TRACE_NONE, Am_INTER_TRACE_ALL,
          Am_INTER_TRACE_EVENTS, Am_INTER_TRACE_SETTING,
          Am_INTER_TRACE_PRIORITIES, Am_INTER_TRACE_NEXT,
          Am_INTER_TRACE_SHORT } Am_Inter_Trace_Options;

void Am_Set_Inter_Trace(); //prints current status
void Am_Set_Inter_Trace(Am_Inter_Trace_Options trace_code);
void Am_Set_Inter_Trace(Am_Object inter_to_trace);
void Am_Clear_Inter_Trace();
```

By default, tracing is off. Each call to Am_Set_Inter_Trace *adds* tracing of the parameter to the set of things being traced (except for Am_INTER_TRACE_NONE which clears the entire trace set). The options for Am_Set_Inter_Trace are:

- no parameters: If Am_Set_Inter_Trace is called with no parameters, it prints out the current tracing status.
- Am_INTER_TRACE_NONE: If Am_Set_Inter_Trace is called with zero or Am_INTER_TRACE_NONE, then it sets there to be nothing be traced. This is the same as calling Am_Clear_Inter_Trace.
- Am_INTER_TRACE_ALL: Traces everything.
- Am_INTER_TRACE_EVENTS: Only prints out the incoming events, and not what happens as a result of these events. When you trace anything else, Amulet automatically also adds Am_INTER_TRACE_EVENTS to the set of things to trace, so you can tell the event which causes things to be updated.
- Am_INTER_TRACE_SETTING: This very useful option just shows which slots of which objects are being set by Interactors and Commands. It is very useful for determining why an object slot is being set.
- Am_INTER_TRACE_PRIORITIES: This prints out changes to the priority levels.
- Am_INTER_TRACE_NEXT: This turns on tracing of the next Interactor to be executed. This is very useful if you don't know the name of the Interactor to be traced.
- Am_INTER_TRACE_SHORT: This prints out only the name of the Interactors which are run.
- an Interactor: This prints lots of information about the execution of that one Interactor.

We have found that tracing an interactor will usually tell you why it didn't run when you expected it to, what it is changing, in case the right objects do not seem to be set, and what interactors are running and why, if wrong ones seem to be running. It is also useful to set breakpoints or traces on object slots using the Inspector to figure out why a slot is being set with incorrect values.



6. Widgets

Amulet provides a full set of widgets, including buttons, menus, scroll bars, and text input fields. Eventually, these will display themselves in different looks, corresponding to the various platforms. The built-in widgets have a large number of parameters to allow programmers to customize them, and the programmer can also create new kinds of widgets by writing new methods.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

6.1 Introduction

Many user interfaces, spanning a wide variety of applications, have several elements in common. Menus and scroll bars, for example, are used so frequently that an interface designer would waste considerable time and effort recreating those objects each time they were required in an application.

The intent of the Amulet Widget set is to supply several frequently used objects that can be easily customized by the designer. By importing these pre-constructed objects into a larger Amulet interface, the designer is able to specify in detail the desired appearance and behavoir of the interface, while avoiding the programming that this specification would otherwise entail.

This document is a guide to using Amulet's Widgets. The objects were constructed using the complete Amulet system, and their descriptions assume that the reader has some knowledge of the components of this system: Opal, Interactors, and ORE.

All of the widgets (except the `Am_Selection_Widget`) are set up to operate with the left mouse button, and ignore the modifier keys. Thus, clicking on a scroll bar with `SHIFT_CONTROL_LEFT_DOWN` does the same as regular left down. The default widget start character is exported as `Am_Default_Widget_Start_Char`.

6.1.1 Current Widgets

Amulet currently supports the following widgets. These widgets are described in this chapter in detail, and summarized in chapter 8.

- `Am_Border_Rectangle`: a rectangle with a raised (or lowered) edge, but no interaction.
- `Am_Button`: a single button
- `Am_Button_Panel`: a panel consisting of multiple buttons with the labels inside the buttons.
- `Am_Checkbox_Panel`: a panel of toggle checkboxes with the labels next to the checkboxes.
- `Am_Radio_Button_Panel`: a panel of mutually exclusively selectable radio buttons with the labels next to the radio buttons.
- `Am_Menu`: a menu panel
- `Am_Menu_Bar`: a menu bar used to select from several different menu panels
- `Am_Option_Button`: a button showing the current choice that pops up a menu of choices.
- `Am_Vertical_Scroll_Bar`: scroll bar for choosing a value from a range of values.
- `Am_Horizontal_Scroll_Bar`: scroll bar for choosing a value from a range of values.
- `Am_Scrolling_Group`: an Amulet group with (optional) vertical and horizontal scrollbars
- `Am_Text_Input_Widget`: a field to accept text input, like for a filename.
- `Am_Selection_Widget`: which supplies the conventional square selection handles around one or more graphical objects and allows them to be moved and grown.

There are four dialog boxes that provide facilities many applications will find useful:

- `Am_Alert_Dialog`: pops up a window that contains a text prompt and an OK button to dismiss the window.
- `Am_Choice_Dialog`: pops up a window that contains a text prompt and allows the user to choose among a set of buttons.
- `Am_Text_Input_Dialog`: pops up a window that contains a text prompt and allows the user to fill in a text input field.
- `Am_Undo_Dialog_Box`: which supports the standard selective undo mechanism and is described in the Interactors chapter, Section 5.6.2.3.2.

In addition, there are a set of command objects that can be put into widgets to perform many of the standard editing operations:

- `Am_Selection_Widget_Select_All_Command`: used with the selection widget to cause everything to be selected ("Select All").
- `Am_Graphics_Set_Property_Command`: to set the color, line style or other property of the selected objects.
- `Am_Graphics_Clear_Command`: delete the selected objects ("Clear")
- `Am_Graphics_Clear_All_Command`: "Clear all"
- `Am_Graphics_Copy_Command`: Copy to the clipboard.
- `Am_Graphics_Cut_Command`: Copy objects to the clipboard and then delete them
- `Am_Graphics_Paste_Command`: Paste a copy of the objects in the clipboard.
- `Am_Undo_Command`: Perform a single undo.
- `Am_Redo_Command`: Perform a single redo.
- `Am_Graphics_To_Bottom_Command`: Make the selected objects be covered by all other objects.
- `Am_Graphics_To_Top_Command`: Make the selected objects be covered by no other objects.
- `Am_Graphics_Duplicate_Command`: Duplicate the selected objects.
- `Am_Graphics_Group_Command`: Make a group out of the selected objects.
- `Am_Graphics_Ungroup_Command`: Ungroup the selected objects.
- `Am_Quit_No_Ask_Command`: Quit the application immediately without asking for confirmation.

6.1.2 Customization

We have tried to make the widgets flexible enough to meet any need. Each widget has a large number of slots which control various properties of its appearance and behavior, which you can set to customize the look and feel. The designer may choose to leave many of the default values unchanged, while modifying only those parameters that integrate the object into the larger user interface.

The visual appearance and the functionality of a widget is affected by values set in its slots. When instances of widgets are created, the instances inherit all of the slots and slot values from the prototype object. The designer can then change the values of these slots to customize the widget. Instances of the custom widget will inherit the customized values. The slot values in a widget prototype can be considered "default" values for the instances.

6.1.3 Using Widget Objects

Include files necessary to use Amulet widgets are `widgets.h` for the widget object definitions, and `standard_slots.h` for the widget slot definitions. These files are included in `amulet.h`, providing a simple way to make sure all needed files are included. Programmers who are designing their own custom widget objects will also need `widgets_advanced.h`. For a complete description of Amulet header files and how to use them most effectively in your project, see Section 1.6 in the Overview chapter.

Widgets are standard Amulet objects, and are created and modified in the same way as any other Amulet object. The following sample code creates an instance of `Am_Button`, and changes the values of a few of its slots.

```
Am_Object my_button = Am_Button.Create("My Button")
  .Set (Am_LEFT, 10)  // set the position of the button
  .Set (Am_TOP, 10)
  .Set (Am_COMMAND,"Push Me");
       // a string in the Am_COMMAND slot specifies the button's label: see below
```

6.1.4 Application Interface

Like interactors, widgets interface to application code through *command objects* added as parts of the widgets. Please see Section 5.6 on Commands in the Interactor's chapter. In summary, instead of executing a call-back procedure as in other toolkits, Amulet widgets call the `Am_DO_METHOD` of the command object stored as the `Am_COMMAND` part of the widget.

6.1.4.1 Accessing and Setting Widget Values

In addition to the `Am_DO_METHOD`, each command also contains the other typical slots of command objects. In particular, the `Am_VALUE` slot of the command object is normally set with the result of the widget. Of course, the type of this value is dependent on the type of the widget. For scroll bars, the value will be a number, and for a checkbox panel, it will be a list of the selected items.

The `Am_VALUE` slot of the widget is also set with the current value of the widget. If you want to set the value of the widget (change the displayed value of the widget), you can set the value of the `Am_VALUE` slot of the widget to the correct value. **Note: Set the Am_VALUE slot of the widget *not* of the command object**, to change the value of the widget. It is also appropriate to put a constraint into the `Am_VALUE` slot of the widget if you want the value shown by the widget to track a slot of another object.

In some situations, the programmer might want to have a constraint dependent on the Am_VALUE slot. This constraint can perform side effects like updating an external data base or even setting slots in Amulet objects or creating or destroying new objects. Other times, the programmer will need to write an Am_DO_METHOD which will typically access the value in the command's Am_VALUE slot. An example of each of these methods can be found below. Of course, if you write your own Am_DO_METHOD and you want the widget to be undo-able, you will also need to write a corresponding Am_UNDO_METHOD, etc. See Section 5.6 in the Interactors chapter on commands for more information.

6.1.4.2 Commands in Widgets

All of the widgets are designed so the command objects are completely replaceable. Thus, you can put the commands from the library or your new commands into any widget. Alternatively, you can get the default commands in the widgets (which has all blank methods) and override the methods. For example:

```
vscroll = Am_Vertical_Scroll_Bar.Create()
  .Set(Am_LEFT, 450)
  .Set(Am_TOP, 80)
  ;
vscroll.Get_Part(Am_COMMAND)
       .Set(Am_DO_METHOD, my_do)
       .Set(Am_UNDO_METHOD, my_undo)
       ;
```

6.1.4.3 Undoing Widgets

Internally, each widget is implemented using graphical objects and interactors. Each internal interactor has its own associated command objects, but these are normally irrelevant to the programmer, since the internal command objects will call the top-level widget command object automatically. This is achieved because the internal commands have their Am_IMPLEMENTATION_PARENT slot of the internal command objects to be the widget's command object, and then Amulet automatically does the right thing.

By default, when commands that are put into widgets are undone, the widget's internal commands are also undone. This means that the widget will typically go back to their original value when the user selects undo, redo or selective undo or repeat.

6.2 The Standard Widget Objects

This section describes the widgets in detail. Each object contains customizable slots, but the designer may choose to ignore many of them in any given application. Any slots not explicitly set by the application are inherited from the widget's prototype.

6.2.1 Slots Common to All Widgets

There are several slots the programmer can set which are used by all widgets in a similar way:

- Am_TOP, Am_LEFT: As with all graphical objects, these slots describe the location of the widget, in coordinates relative to the object's parent's location. Default values are 0 for both top and left.
- Am_VISIBLE: If this boolean is true, the object is visible; otherwise, it is not drawn on the screen. Default is true.
- Am_VALUE: The current value computed by the widget. This slot can also be set to change the widget's value.
- Am_WIDGET_LOOK: The value of this slot tells Amulet how you want your widgets to look when drawn on the screen. Possible values are Am_MOTIF_LOOK, Am_WINDOWS_LOOK, or Am_MACINTOSH_LOOK. Any look will eventually be available on any platform, but currently only Am_MOTIF_LOOK is implemented.
- Am_FILL_STYLE: This slot determines the color of the widget. Amulet automatically figures out appropriate foreground, background, shadow, and highlight colors given a fill color. Acceptable values are any Am_Style, and the default is Am_Amulet_Purple. The only part of the style used is the color of the style. On a black and white screen, a default set of stipples are used to make sure the widgets are visible.
- Am_ACTIVE_2: This slot turns off interaction with the widget without turning it grey. This is mainly aimed at interactive tools like Interface Builders that want to allow users to select and move widgets around. It might also be useful in a multi-user situation where users who do not have the "floor" should not have their widgets responding. For a widget to operate, both Am_ACTIVE_2 and Am_ACTIVE must be true. The default value is true.

The command objects in all widgets have the following standard slots:

- Am_ACTIVE: This slot in the command is used to determine whether the widget is enabled or not (greyed out). Often, this slot will contain a formula dependent on some system state. The default value is true. (Actually, the widget itself also contains an Am_ACTIVE slot, but this one should not normally be used. The widget-level slot contains a constraint that depends on the Am_ACTIVE slot of the command object part of the widget.)
- Am_VALUE: This slot is set to the current value of the widget. Do *not* set this slot in the command object to try to change the widget's value (see the Am_VALUE of the widget instead).
- Am_DO_METHOD: The method to be called when the widget executes. This procedure takes one parameter: the command object.
- All the various undo, redo and selective methods, as described in Section 5.6.2.3 of the Interactors chapter.
- Am_IMPLEMENTATION_PARENT: If you want the command to invoke another command, you can set this slot in the widget's command to the other command object. For example, if the widget is the "OK" button of a dialog box, the Am_IMPLEMENTATION_PARENT of the OK widget's command might be the command object for the entire dialog box. Then Amulet will correctly know how to handle Undo, and it will call the parent command automatically.

6.2.2 Border_Rectangle

The Am_Border_Rectangle has a raised or lowered edge of a lighter or darker shade of the Am_FILL_STYLE. It ignores the Am_LINE_STYLE. It looks pressed in if Am_SELECTED is true, and sticking out of the screen if Am_SELECTED is false. This widget has no interaction or response to the mouse.

Slot	Default Value	Type	
Am_SELECTED	false	bool	
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	*{Am_MOTIF_LOOK, Am_MACINTOSH_LOOK, Am_WINDOWS_LOOK }*
Am_WIDTH	50	int	
Am_HEIGHT	50	int	
Am_TOP	0	int	
Am_LEFT	0	int	
Am_VISIBLE	true	bool	
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style	

6.2.3 Buttons and Menus

All of the buttons and menus operate fairly similarly.

For a single, stand-alone button, the Am_COMMAND slot can either be the string or object to display in the button, or it can be an Am_COMMAND object, in which case, the label of the widget is determined by the Am_LABEL slot of the Am_COMMAND part of the widget, which itself should be a string or object, as described below.

The various panel objects (that display a set of buttons) and the menus (that display a set of buttons) all take an Am_ITEMS slot which must contain an Am_VALUE_LIST. The items in this value list can be:

- a C string (char*), in which case this string is displayed as the label,
- a graphical object, in which case this object (or an instance of this object if the object is already a part of another object) is displayed as the label. This object can of course be a group, so arbitrary pictures can be displayed as the value of a widget.
- a command object, in which case the value of the Am_LABEL field of the command object is used as the item's label. The Am_LABEL field itself can contain either a C string or a graphical object.

There are two basic ways to use the panel-type objects, including menus:

1. Each individual item has its own command object, and the Am_DO_METHOD of this command does the important work of the item. This would typically be how menus of

operations like Cut, Copy, and Paste would be implemented.

2. The top-level panel itself has a command object and the individual items do not have a command object. For example, the Am_ITEMS slot of the widget contains an Am_VALUE_LIST of strings. In this case, the top-level command object's Am_DO_METHOD will be called, and it typically will look in its Am_VALUE slot to determine which item was selected. This method is most appropriate when the panel or menu is a list of values, like colors or fonts, and you do not want to create a command for each item.

Note that the top-level command object is not called if the individual item has a command object, unless you explicitly set the Am_IMPLEMENTATION_PARENT of the item's command to be the widget's command. It would be unusual, but is perfectly legal, to have a Am_Value_List that contains some commands and some strings.

6.2.3.1 Commands in Buttons and Menus

Slots of the command object used by buttons and menus are as follows. More details are available in Section 5.6 of the Interactor chapter.

- Am_LABEL: This slot can contain a string or a graphical object, which will be drawn as the label for this item.
- Am_ID: Normally, buttons set the Am_VALUE slot to the Am_LABEL of the command. However, this typically requires doing string matching. Therefore, if the Am_ID field is non-zero, then the Am_VALUE slot is set with the value of the Am_ID slot instead of the Am_LABEL slot. The Am_ID slot can contain any type of value.
- Am_ACTIVE: This controls whether the widget is active or not (greyed out). If the Am_ACTIVE slot of the top-level command in a panel or menu is set to false, then all the items are greyed out. More typically, the Am_ACTIVE slot of the command associated with a single item will be `false`, signaling that just that one item should be greyed out.
- Am_ACCELERATOR: For menus, menu-bars and option-buttons, this can contain an Am_Input_Char or a string representing the character, and will make that character be an accelerator for this command in the window. See Section 6.2.3.2 for more details.
- Am_VALUE: This slot is set by the widget with the label(s) or ID(s) of currently the selected button(s). In a single button, this contains 0 if the button is not selected, or the button's label or ID if it is selected. Thus, you can use Valid() to determine if the button is pressed. If multiple items can be selected, as in a check box panel or for a button panel if you set Am_HOW_SET to be Am_CHOICE_LIST_TOGGLE, then this slot will always contain an Am_Value_List with the labels or IDs of the selected items. If no items are selected, then the list will be empty. Note: the value of the Am_VALUE slot will *not* be 0; it will be a list that is empty, so you cannot use Valid() since an empty Am_Value_List is still valid. For panels where only a single item can be selected, such as a radio button panel or button panels with Am_HOW_SET set to be Am_CHOICE_SET, the Am_VALUE slot is set to the single button's label or ID, or 0 if nothing is selected.

6.2.3.2 Accelerators for Commands

If the Am_ACCELERATOR slot of a Command object for a menu item is set with an Am_Input_Char (see Section 5.3.3.1) or a string that can be converted into a character description, like "CONTROL_F7", then Amulet will automatically create an accelerator for that command in the window the widget is attached to. In addition, the menu item will show the accelerator using the "short string" form of Am_Input_Chars. This accelerator will only be active when the command is active.

The interface to the accelerator lists for windows is available in widget_advanced.h through the functions Am_Add_Accelerator_Command_To_Window and Am_Remove_Accelerator_Command_From_Window, which take the window and command object.

6.2.3.3 Am_Menu_Line_Command

Am_Menu_Line_Command is a special purpose type of command object provided by Amulet to draw horizontal dividing lines in menus. To add a horizontal line in a menu, simply include an instance of Am_Menu_Line_Command in the menu's Am_ITEMS list. An example of this can be found in section 6.2.3.7. Am_Menu_Line_Command has no customizable slots, and it is an inactive menu item.

Slot	Default Value	Type
Am_LABEL	"Menu_Line_Command"	Am_String
Am_ACTIVE	false	bool
Am_VALUE	NULL	Am_Value

6.2.3.4 Am_Button

The Am_Button object is a single stand-alone button. A button can have a text label, or can contain an arbitrary graphical object when drawn.

Slot	Default Value	Type	
Am_VALUE	NULL	Am_Value	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	<formula>	int	
Am_HEIGHT	<formula>	int	
Am_H_ALIGN	Am_CENTER_ALIGN	{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}	
Am_FIXED_WIDTH	Am_NOT_FIXED_SIZE	int	
Am_FIXED_HEIGHT	Am_NOT_FIXED_SIZE	int	
Am_INDENT	0	int	
Am_MAX_RANK	false	bool	
Am_MAX_SIZE	false	bool	
Am_ITEM_OFFSET	5	int	
Am_ACTIVE	<formula>	bool	
Am_ACTIVE_2	true	bool	
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	*{Am_MOTIF_LOOK, Am_MACINTOSH_LOOK, Am_WINDOWS_LOOK}*
Am_KEY_SELECTED	false	bool	
Am_FONT	Am_Default_Font	Am_Font	
Am_FINAL_FEEDBACK_WANTED	false	bool	
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style	
Am_COMMAND	Am_Command	Am_Command	

Special slots of Am_BUTTONS are:

- Am_WIDTH, Am_HEIGHT: By default, the width and height of the button are automatically calculated by formulas in these slots. A button is made big enough to contain its text label or graphical object, including borders, and offset pixels (see below). A user can replace the width and height formulas by setting these slots directly. Once the values are set with new values or formulas, the formulas will be removed.
- Am_ITEM_OFFSET: The string or object displayed inside the button is set away from the border of the button by Am_ITEM_OFFSET pixels, in both the horizontal and vertical directions. The default is 5.
- Am_FONT: The button's text label (if any) is drawn in this font. Acceptable values are any Am_Font, and the default is Am_Default_Font.

- Am_FINAL_FEEDBACK_WANTED: This determines if the button should be drawn as if it is still selected, even after user interaction has stopped. This is useful if you want to use the button to show whether it is selected or not. The default is false.

6.2.3.5 Am_Button_Panel

An Am_Button_Panel is a panel of Am_Buttons, with a single interactor in charge of all the buttons. Since an Am_Button_Panel's prototype object is a Am_Map, all the slots that Am_Map uses are also used by Am_Button_Panel. See the Opal chapter for a description of Am_Map. Some Am_Map slots are described below along with slots specific to Am_Button_Panel.

Slot	Default Value	Type	
Am_VALUE	NULL	Am_Value	
Am_WIDTH	Am_Width_Of_Parts	int	
Am_HEIGHT	Am_Width_Of_Parts	int	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	<formula>	int	*Read-only*
Am_HEIGHT	<formula>	int	*Read-only*
Am_HOW_SET	Am_CHOICE_SET	Am_How_Set	
Am_ITEM_OFFSET	3	int	
Am_ACTIVE	<formula>	bool	
Am_ACTIVE_2	true	bool	
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	*{Am_MOTIF_LOOK, Am_MACINTOSH_LOOK , Am_WINDOWS_LOOK}*
Am_KEY_SELECTED	false	bool	
Am_FONT	Am_Default_Font	Am_Font	
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style	
Am_FINAL_ FEEDBACK_WANTED	false	bool	
Am_LAYOUT	Am_Vertical_ Layout	{Am_Vertical_Layout, Am_Horizontal_Layout, NULL, etc.}	
Am_H_ALIGN	Am_LEFT_ALIGN	{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}	
Am_ITEMS	0	int, Am_Value_List of commands or strings, etc.	
Am_COMMAND	Am_Command	Am_Command	

Special slots of Am_Button_Panel are:

- Am_FONT: The font used for the button labels.
- Am_FINAL_FEEDBACK_WANTED: Whether to show which item is selected or not. The default is false.
- Am_WIDTH, Am_HEIGHT: The width and height slots contain the standard Amulet formulas Am_Width_Of_Parts and Am_Height_Of_Parts, respectively. If these slots are set to specific values by the designer, those values replace the formulas, and the panel will no longer resize itself if its contents change.
- Am_FIXED_WIDTH: This slot determines how the buttons in a panel get their width. An integer value of 0, or a boolean value of false, means each button is as wide as its calculated width based on the contents. Thus, each button will be a different size. An integer value of 1, or a boolean value of true, means that all the buttons in the panel are set to be as wide as the calculated width of the widest button in the panel. An integer value greater than 1 sets the width of all buttons in the panel to that specific value. The default is true.
- Am_FIXED_HEIGHT: This slot determines the height of buttons in a panel. It acts the same way as Am_FIXED_WIDTH. The default is false.
- Am_HOW_SET: This slot determines whether single or multiple buttons can be selected. Its default value is Am_CHOICE_SET, which allows a single selection. Changing this to Am_CHOICE_TOGGLE will allow the selected item to be turned off by clicking it again. Am_CHOICE_LIST_TOGGLE allows there to be multiple selections. See the Interactors manual for a complete description of the legal values.
- Am_LAYOUT: This specifies a function determining how the button panel should be arranged. A more complete description of the slot can be found in section 4.7.2. Am_Vertical_Layout is the default, and Am_Horizontal_Layout is another good value.
- Am_H_ALIGN: In a vertically arranged button panel with variable width buttons, this determines how the buttons should be arranged in the panel. The default is Am_LEFT_ALIGN, and other possible values are Am_CENTER_ALIGN and Am_RIGHT_ALIGN.
- Am_V_ALIGN: This slot works like Am_H_ALIGN, except is only used in horizontally arranged panels width variable height buttons. Possible values are Am_TOP_ALIGN, Am_CENTER_ALIGN, and Am_BOTTOM_ALIGN.
- Am_ITEMS: This slot specifies the items which are to be put in the button panel. An Am_Value_List should be used to specify specific items to add to the panel. See section 6.2.3.1 for a complete description. In summary, elements of the value list can be either strings, graphical objects, or command objects. A string value is used as the label for the button in the panel. A graphical object is displayed in the button. A command object is used to specify a custom command for that particular button in the panel. For commands, the button's string label or graphical object is taken from the command object's Am_LABEL slot.
- Am_COMMAND: This slot contains an Am_Command object. See section 6.2.3.1 for a complete description.

6.2.3.5.1 Example of Using a Button Panel

Each button in the panel is drawn with a text label or a graphical object inside it. An Am_Value_List in the Am_ITEMS slot tells the button panel what to put inside each button. If a string is specified, it is used as the button's label. If a graphical object is specified, it is drawn in the button. If a command object is specified, that command object's Am_DO_METHOD method is called each time the button is pressed, and the button's label or graphical object is obtained from the command object's Am_LABEL slot. The following code specifies a button panel with three buttons in it.

```
// a graphical object and custom do action, defined elsewhere:
extern Am_Object My_Graphical_Object;
extern void My_Custom_Do_METHOD (Am_Object command_obj);
Am_Object my_command;
// my button panel:
Am_Object My_Button_Panel = Am_Button_Panel.Create ("My Button Panel")
  .Set (Am_ITEMS,
        Am_Value_List ()
          .Add ("Push me.")
          .Add (My_Graphical_Object)
          .Add (my_command = Am_Command.Create()
                  .Set (Am_LABEL, "Push me too.")
                  .Set (Am_DO_METHOD, My_Custom_Do_Method)));
```

The first button in the panel is drawn with the text label "Push me." and does not have its own command object. The second button in the panel is drawn containing My_Graphical_Object drawn inside it, and also does not have its own command. The third button in the panel is drawn with the text label "Push me too." and has its own command object associated with it.

When the third button is pressed, My_Custom_Do_Method is called, with the button's command object (my_command) as an argument. The command object's Am_VALUE slot will already have been set with either 0, if the button was not selected, or "Push me too." if the button was selected. We assume that this command is not undoable since there is no custom Undo action to go with My_Custom_Do_Method.

If any of the other buttons in the panel are pressed, the do action of My_Button_Panel's command object (in its Am_COMMAND slot) will be called, with the command object as an argument. The Am_VALUE of the command object is set with the labels or objects corresponding to the currently selected buttons.

If you wanted the button panel's command to be invoked when the third button was pressed, you would have to set the third button's command object's Am_IMPLEMENTATION_PARENT slot to contain the button panel's command object. For example, after executing the following code, My_Other_Custom_Do_Method in the panel will be called when *any* of the buttons are selected.

```
Am_Object panel_command = My_Button_Panel.Get(Am_COMMAND);
panel_command.Set (Am_DO_METHOD, My_Other_Custom_Do_Method);
my_command.Set (Am_IMPLEMENTATION_PARENT, panel_command);
```

6.2.3.6 Am_Radio_Button_Panel

A radio button panel is a set of small buttons with items appearing either to the right or left of each button. Exactly one button from the set can be selected at any particular time, and the button stays selected after the user stops interacting with it. The radio button panel is often used to present a user with several different options, only one of which can be in effect at any particular time.

Slot	Default Value	Type
all the slots of the button panel, with the following changes		
Am_BOX_WIDTH	15	int
Am_BOX_HEIGHT	15	int
Am_BOX_ON_LEFT	true	bool
Am_FIXED_WIDTH	false	int, bool
Am_FINAL_FEEDBACK_WANTED	true	bool
Am_H_ALIGN	<formula>	{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}

An Am_Radio_Button_Panel is essentially the same as an Am_Button_Panel, with a few exceptions. There are a few new slots, and some of the defaults of the other slots are different. All other slots not listed below act the same way as in an Am_Button_Panel. Since radio buttons always only allow a single selection, the Am_VALUE slot of the top-level Am_COMMAND is always set with either 0 or the ID or label of the selected item.

- Am_BOX_HEIGHT, Am_BOX_WIDTH: These specify the size in pixels of the small radio button box that is drawn next to the item in the button. The defaults are 15 for each.
- Am_BOX_ON_LEFT: This boolean determines whether the radio box should be drawn to the left of the item, or to the right. If true, the box is drawn on the left, and if false, it is drawn on the right. The default is true.
- Am_H_ALIGN: This slot contains a formula which evaluates to Am_LEFT_ALIGN if the Am_BOX_ON_LEFT is true, or Am_RIGHT_ALIGN if it is false.
- Am_FIXED_WIDTH: The default is false for this slot in a radio button panel.
- Am_FINAL_FEEDBACK_WANTED: The default is true for this slot in a radio button panel. This makes sure the user selected button stays selected after interaction is complete.
- Am_HOW_SET: Since radio buttons are only allowed to have a single selection, this slot defaults to Am_CHOICE_SET. However, if you want to allow the user to turn off the current selection by clicking on it again, you can set this slot to be Am_CHOICE_TOGGLE. It would be wrong to use Am_CHOICE_LIST_TOGGLE.

6.2.3.7 Am_Checkbox_Panel

A checkbox panel is a set of small buttons with items appearing either to the right or left of each button. Zero or more buttons from the set can be selected at any particular time, and the buttons stay selected after the user stops interacting with the panel. The checkbox panel is often used to present a user with several different options that can be in effect at the same time.

Slot	Default Value	Type
all the slots of the button panel, with the following changes		
Am_HOW_SET	Am_CHOICE_ LIST_TOGGLE	Am_How_Set
Am_BOX_WIDTH	15	int
Am_BOX_HEIGHT	15	int
Am_BOX_ON_LEFT	true	bool
Am_FIXED_WIDTH	false	int, bool
Am_FINAL_FEEDBACK_WANTED	true	bool
Am_H_ALIGN	<formula>	{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}

An Am_Checkbox_Panel is essentially the same as an Am_Radio_Button_Panel. It is drawn slightly differently, and the following slot is different:

- Am_HOW_SET: The default for a checkbox panel is Am_CHOICE_LIST_TOGGLE, which allows multiple items to be selected at the same time.

Since multiple items can be selected in an Am_Checkbox_Panel, the Am_VALUE slot of the top-level Am_COMMAND contains an Am_VALUE_LIST of the labels or IDs of the selected items.

6.2.3.8 Am_Menu

An Am_Menu is a single menu panel, implemented as another form of Am_Button_Panel. A menu panel has a background rectangle behind it, and the items are drawn differently than in Am_Buttons.

Slot	Default Value	Type
all the slots of the button panel		
Am_FINAL_FEEDBACK_WANTED	false	bool
Am_WIDTH	<formula>	int
Am_HEIGHT	<formula>	int
Am_X_OFFSET	2	int
Am_Y_OFFSET	2	int
Am_V_SPACING	-2	int

The following slots of an Am_Menu differ from those in an Am_Button_Panel:

- Am_WIDTH, Am_HEIGHT: The default formulas in these slots calculate the width and height of the menu's items, and add enough width and height to contain the menu's outer border.
- Am_HOW_SET: The default for a menu is Am_CHOICE_SET.
- Am_X_OFFSET, Am_Y_OFFSET: These slots cause the menu items to be offset from the upper left corner of the menu. The defaults are 2 for X and Y, to make room for the outer border around the menu.
- Am_V_SPACING: This creates extra space between the items in a menu. The default value is -2, which pushes the menu items vertically closer together in the menu.
- Am_TEXT_OFFSET: This offset is used only in the horizontal direction when a text label is being displayed in the menu item (as opposed to a graphical object). This allows greater horizontal spacing for text, while keeping the standard vertical spacing. The default value is 2 pixels.

6.2.3.8.1 Simple Example

Here is an example of creating an Am_Menu object.

```
Am_Object my_menu = Am_Menu.Create("my_menu")
   .Set (Am_LEFT, 150)
   .Set (Am_TOP, 200)
   .Set (Am_ITEMS, Am_Value_List()
            .Add ("Menu item")
            .Add (Am_Menu_Line_Command.Create("my menu line"))
            .Add (Am_Command.Create("item2")
                   .Set(Am_ACTIVE, false)
                   .Set(Am_LABEL, "Not active"))
            .Add (Am_Command.Create("item2")
                   .Set(Am_LABEL, ("Active item"))
                   .Set(Am_ACCELERATOR, "CONTROL_a"));
my_window.Add_Part(my_menu);
```

The menu has three menu items with a line between the first and second items. The first item appears as "Menu item" in the menu, and has no corresponding command object. If that item is selected by the user, the do action of my_menu's command object will be called with "Menu item" in its Am_VALUE slot. Since there is no command object associated with the first menu item, there is no way to make it inactive without making the whole menu inactive.

The second menu item appears in the menu as "Not active". It will be grayed out, because the Am_ACTIVE slot of its corresponding button command object is set to false. This item cannot be chosen from the menu because it is inactive.

The third menu item appears in the menu as "Active item ^a". It does have a command object associated with it, so if it is selected by the user, that command's do action will be executed, and the widget's top level command will not be executed. The widget's top level command object is not called unless you set the individual button command object's Am_IMPLEMENTATION_PARENT slot to point to it. This command has an accelerator, so if the user hits control-a in my_window, the command will be executed.

6.2.3.9 Am_Menu_Bar

The Am_Menu_Bar is a menubar like you might find at the top of a window that has a horizontal row of items you can select, and each one pops down a menu of further options. Sometimes it is called a pull-down menu. Amulet's menu bar currently supports a single level of sub-menus (no pull-outs from the pull-downs). However, any menu item (either at the top level or a sublevel) can be an abitrary Amulet object, just like with other button-type objects.

Slot	Default Value	Type	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	<formula>	int	*width of owner*
Am_HEIGHT	<formula>	int	*height of text in menubar*
Am_ACTIVE	<formula>	bool	
Am_ACTIVE_2	true	bool	
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	
Am_FONT	Am_Default_Font	Am_Font	
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style	
Am_ITEMS	NULL	Am_Value_List	
Am_COMMAND	Am_Command	Am_Command	

The interface to menu bars is similar to other button widgets: the Am_ITEMS slot of the menu_bar object should contain an Am_Value_List. However, unlike other objects, the list *must* contain command objects. The label field of this command object serves as the top-level menubar item. In the command object should be an Am_ITEMS slot containing an Am_Value_List of the sub-menu items. This list can contain command objects, strings or Amulet objects, as with other menus and button panels. For example:

```
my_menu_bar = Am_Menu_Bar.Create()
  .Set(Am_ITEMS, Am_Value_List ()
        .Add (Am_Command.Create("File_Command")
              .Set(Am_LABEL, "File")
              .Set(Am_DO_METHOD, my_file_do)
              .Set(Am_ITEMS, Am_Value_List ()
                    .Add ("Open...")
                    .Add ("Save As...")
                    .Add (Am_Command.Create("Quit_Command")
                          .Set(Am_DO_METHOD, my_quit)
                          .Set(Am_LABEL, "Quit")
                          .Set(Am_ACCELERATOR, "^q"))
                    )
              )
        .Add (Am_Command.Create("Edit_Command")
              .Set(Am_LABEL, "Edit")
              .Set(Am_DO_METHOD, my_edit_do)
              .Set(Am_ITEMS, Am_Value_List ()
                    .Add (undo_command.Create())
                    .Add ("Cut")
                    .Add ("Copy")
                    .Add ("Paste")
                    .Add (Am_Menu_Line_Command.Create("my menu line"))
```

```
                    .Add ("Find...")
                    )
                )
            )
```

If a sub-menu item has a command (like Quit or Undo above), then its Am_DO_METHOD is called when the item is chosen by the user. If it does *not* have a command object (like Cut and Paste above), then the command object of the main item is used (here, the do method called my_edit_do in the command object named Edit_Command will be called for Cut and Paste, and the Am_VALUE slot of the Edit_Command will be set to the string of the particular item selected). Note that because the first level value list must contain command objects, the command object stored in the menu_bar object itself will never be used unless the programmer explicitly sets the Am_IMPLEMENTATION_PARENT slot of a command to the menu_bar's command object. The Am_VALUE of whatever command object is executed will be set to the label or ID of the selected item. Menu bars can also contain accelerators, as shown by the Quit command in the example.

Am_Menu_Bars allow the top level item to be chosen (unlike, say the Macintosh), in which case its command object is called with its own label or ID as the Am_VALUE. The programmer should ignore this selection if, as usually is the case, pressing and releasing on a top-level item should do nothing.

Individual items can be made unselectable by simply setting the Am_ACTIVE field of the command object for that item to false. If the Am_ACTIVE field of a top-level command object is false, then the entire sub-menu is greyed out, although it will still pop up so the user can see what's in it.

Unlike regular menus and panels, the Am_Menu_Bar will not show the selected value after user lets up with the mouse. That is, you cannot have Am_FINAL_FEEDBACK_WANTED as true.

Slots that behave different for the Am_Menu_Bar are:

- Am_WIDTH, Am_HEIGHT: By default, these slots contain formulas that make the menubar be the width of its owner (usually the width of the window) and the height of the current font. However, you can override these defaults with constant values or other formulas.

6.2.3.10 Am_Option_Button

An Option Button is a widget that acts like a menu, but displays only the current value. It looks like a button, with a little notch at the right, and when the user clicks on it, a menu pops up listing all the choices. If the user releases, the value does not change, but if the user moves the mouse and releases, the new selection is shown in the button. The parameters to an Am_Option_Button are the same as those to a Menu, except that Am_FINAL_FEEDBACK_WANTED is ignored, and Am_HOW_SET must stay Am_CHOICE_SET.

6.2.4 Scroll Bars

Am_Vertical_Scroll_Bar and Am_Horizontal_Scroll_Bar are widget objects that allow the selection of a single value from a specified range of values, either as a int or a float (see section 6.2.4.1). You specify a minimum and maximum legal value, and the scroll bar allows the user to pick any number in between. The user can click on the indicator and drag it to set the value. As the indicator is dragged, the value is updated continuously. If the user clicks on the arrows, in the scroll bar, the scroll bar increments or decrements the current value by Am_SMALL_INCREMENT. If the user clicks above or below the scroll bar, the value jumps by Am_LARGE_INCREMENT. Unfortunately, auto-repeat (repeatedly incrementing while the mouse button is held down) is not implemented yet. You can also adjust the indicator's size to show what percent of the entire contents is visible.

Like all other widgets, the Am_Vertical_Scroll_Bar and Am_Horizontal_Scroll_Bar store the value in the Am_VALUE slot of the widget and in the Am_VALUE slot of the Am_COMMAND object. As the value is changed by the user, the Am_DO_METHOD of the command is also continuously called. The Am_VALUE slot **of the widget** can also be set by a program to adjust the position of the scroll bar indicator.

The Am_Scrolling_Group provides a convenient interface for a scrollable area. It operates similarly to a regular Am_Group (see the Opal chapter), except that it optionally displays two scroll bars which the user can use to see different parts.

6.2.4.1 Integers versus Floats

There are four slots that control the operation of the scroll bars: Am_VALUE_1, Am_VALUE_2, Am_SMALL_INCREMENT, and Am_LARGE_INCREMENT. If all of these slots hold values of type integer, then the result stored into the Am_VALUE slot will also be an integer. If any of these values is a float, however, then the result will be a float. The default values are 0, 100, 1 and 10, so the default result is an integer. Note that the Inspector and cout display floats without decimal parts as integers, but the scroll bar still treats them as floats.

6.2.4.2 Commands in Scroll Bars

The command objects in scroll bars work similarly to other widget commands. The main difference is in the Am_VALUE slot.

- Am_ACTIVE: This determines whether the scroll bar is active or not. Inactive scroll bars do not respond to user input. However, in the default Motif look and feel, they are *not* drawn any differently.
- Am_VALUE: This holds the currently selected value on the scroll bar. As discussed above, it will either be an integer or float value.

The scroll bar command supplies undo methods which resets the Am_VALUE slot and the displayed value to the previous value. However, most applications do not allow scrolling operations to be undone, in which case, you should make sure that the scrolling command is not queued on the undo list (see the section on Undo in the Interactors manual).

For scrolling groups, the default is not to be undoable. If you want the scrolling group commands to be queued for undoing, set the Am_COMMAND slot of the scrolling group to be NULL.

6.2.4.3 Horizontal and vertical scroll bars

These are the default slots of Am_Vertical_Scroll_Bar. An Am_Horizontal_Scrollbar has the same defaults, except it's 200 pixels wide and 20 pixels high.

Slot	Default Value	Type	
Am_VALUE	50	Am_Value	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	20	int	
Am_HEIGHT	200	int	
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style	
Am_VALUE_1	0	int or float	*Value at top*
Am_VALUE_2	100	int or float	*Value at bottom*
Am_SMALL_INCREMENT	1	int or float	*When click arrow*
Am_LARGE_INCREMENT	10	int or float	*When click "page"*
Am_PERCENT_VISIBLE	0.2	float	*Size of indicator*
Am_COMMAND	Am_Command	Am_Command	

Here is a description of the customizable slots of a scroll bar:

- Am_WIDTH: This determines the width of the scroll bar. This includes the height of the arrows at the ends of horizontal scroll bars. The default is 20 for vertical bars, and 200 for horizontal bars.
- Am_HEIGHT: This determines the height of the scroll bar. This includes the height of the arrows at the ends of vertical scroll bars. The default is 200 for vertical bars, and 20 for horizontal bars.
- Am_VALUE_1: This is the value selected in the scroll bar when the indicator is at the top (for vertical scroll bars) or left (for horizontal) end of the scroll bar. The default type is an int, but it can also be a float. The default is 0. Note that Am_VALUE_1 is not required to be less than Am_VALUE_2, in case you want the bigger value to be at the top or left.
- Am_VALUE_2: This is the value selected in the scroll bar when the indicator is at the bottom (for vertical scroll bars) or right (for horizontal) end of the scroll bar. The defauls type is an int, but it can also hold a float. The default is 100. Note that Am_VALUE_2 is not required to be bigger than Am_VALUE_1.

- Am_SMALL_INCREMENT: This is the amount the value is changed when the user clicks on the arrows at the end of the scroll bar. The default value is 1 of type int. The slot can contain either an int or a float.
- Am_LARGE_INCREMENT: This is the amount the value is changed when the user clicks on the scroll area on either side of scroll handle. The default value is 10 of type int. The slot can contain either an int or a float.
- Am_PERCENT_VISIBLE: This slot specifies how large the indicator will be with respect to the region it is dragged back and forth in. The slot should hold a float between 0.0 and 1.0, and the default is 0.2. If this value determines a thumb smaller than 6 pixels long, a 6 pixel thumb is drawn instead.

6.2.4.4 Am_Scrolling_Group

An Amulet scrolling group is useful when you want to display something bigger than will fit into a window and allow the user to scroll around to see the contents. You can use the Am_Scrolling_Group just like a regular group, but the user will be able to scroll around using the optional vertical and horizontal scrollbars.

Slot	Default Value	Type	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	150	int	
Am_HEIGHT	150	int	
Am_X_OFFSET	0	int	*Where scrolled to*
Am_Y_OFFSET	0	int	*Where scrolled to*
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style	
Am_INNER_FILL_STYLE	0	Am_Style or 0	*If 0 uses FILL_STYLE*
Am_LINE_STYLE	Am_Thin_Line	Am_Style	*Border of scrolling area*
Am_H_SCROLL_BAR	true	bool	*Whether show horiz bar*
Am_V_SCROLL_BAR	true	bool	*Whether show vertical bar*
Am_H_SCROLL_BAR_ON_TOP	false	bool	
Am_V_SCROLL_BAR_ON_LEFT	false	bool	
Am_H_SMALL_INCREMENT	10	int	
Am_H_LARGE_INCREMENT	<formula>	int	*Computed from page size*
Am_V_SMALL_INCREMENT	10	int	
Am_V_LARGE_INCREMENT	<formula>	int	*Computed from page size*
Am_INNER_WIDTH	400	int	*Size of scrollable area*
Am_INNER_HEIGHT	400	int	*Size of scrollable area*

A scrolling group has two distinct rectangular regions. One is the region that is drawn on the screen, and contains scroll bars, and a rectangle with a visible portion of the group. This region is defined by the Am_LEFT, Am_TOP, Am_WIDTH and Am_HEIGHT of the Am_Scrolling_Group itself. The other region is called the inner region which is the size of all the objects, some of which might not be visible. This area is controlled by the Am_INNER_WIDTH and Am_INNER_HEIGHT slots.

By default, the Am_ACTIVE slots of the scroll bars are calculated based on whether the scroll bars are needed (whether any of the group is hidden in that direction). The percent-visible is also calculated based on the amount of the group that is visible. The Am_H_LARGE_INCREMENT and Am_V_LARGE_INCREMENT are also calculated based on the screen size.

6.2.4.4.1 Members of a Am_Scrolling_Group

You can add and remove members to a scrolling group using the regular Add_Part and Remove_Part methods (be sure to adjust the inner size of the group if the new members change it--you can arrange for this to happen automatically by putting an appropriate constraint into the Am_INNER_WIDTH and Am_INNER_HEIGHT slots, such as Am_Width_Of_Parts and Am_Height_Of_Parts.). However, when enumerating the parts of a Am_Scrolling_Group, do *not* use a Am_Part_Iterator, since this will also list the scroll bars. Instead, use the Am_Value_List stored in the Am_GRAPHICAL_PARTS slot of the group, which will only contain the objects you added. The Am_GRAPHICAL_PARTS slot can also be used for normal groups (instances of Am_Group and Am_Map), so you can write code that will operate on either scrolling groups or regular groups.

6.2.4.4.2 Am_Scrolling_Group Slots

- Am_WIDTH, Am_HEIGHT: These default to 150 in a scrolling group. The width and height determine the size of the group's graphical appearance on the screen, including space for scroll bars.
- Am_X_OFFSET, Am_Y_OFFSET: These are the coordinates of the visual region, in relation to the origin of the inner region. The slots always contain a nonnegative integer, with 0 corresponding to no offset (meaning that the scrollable region's top and left are the same at the top, left of the visible area). The default is 0 for both X and Y offset. You may also Get or Set these slots, and the slots can even contain formulas. Getting the slot gives you the current scrollbar position, and setting the slot changes the current scrollbar position and scrolls the area.
- Am_FILL_STYLE: The filling style (color) used to draw the scroll bars (and the background of the window if Am_INNER_FILL_STYLE is 0).
- Am_INNER_FILL_STYLE: This determines what the background fill of the group will be. If it is an Am_Style, that style is used. If it contains 0, the Am_FILL_STYLE slot is used.
- Am_LINE_STYLE: This determines what the border of the scrolling region. The default is Am_Thin_Line. The size of the scrolling area is adjusted inward to make room for the specified line thickness. If 0, then no line is drawn.
- Am_H_SCROLL_BAR, Am_V_SCROLL_BAR: These booleans determine whether the group will have vertical and/or horizontal scroll bars. These slots default to true.

- Am_H_SCROLL_BAR_ON_TOP, Am_V_SCROLL_BAR_ON_LEFT: These booleans determine which side of the group the scroll bars appear on. The defaults are false for both, which puts the horizontal scroll bar at the bottom of the group, and the vertical scroll bar at the right of the group as on most standard windows.
- Am_H_SMALL_INCREMENT, Am_V_SMALL_INCREMENT: This is the small increment in pixels of the horizontal and vertical scroll bars. The value determines how much the scrolling group is moved when the user clicks on the scroll arrows. The default is 10 pixels.
- Am_H_LARGE_INCREMENT, Am_V_LARGE_INCREMENT: This is the large increment, in pixels, of the horizontal and vertical scroll bars. The value determines how much the scrolling group is moved when the user clicks on the scroll areas beside the scroll indicators. The default is calculated by a formula to jump one visible screen full.
- Am_INNER_WIDTH, Am_INNER_HEIGHT: This is the size of the entire group, not just the visible portion. The defaults are 400 for both. This will usually be calculated by a formula based on the contents of the scrolling group (e.g., Am_Width_Of_Parts and Am_Height_Of_Parts). It is OK if these are smaller than the scrolling group's Am_WIDTH and Am_HEIGHT: this just means that the entire area is visible, and so the appropriate scroll bars will be disabled.
- Am_COMMAND: You rarely will need to call a method when the scrolling group is used, so the Am_COMMAND slot has the value Am_NOT_USUALLY_UNDONE. You can change it to NULL to get the internal scrolling operations queued for undo, or you can supply a command to be executed after each scrolling operation.

6.2.4.4.3 Using a Scrolling Group

To use an Am_Scrolling_Group, simply create an instance of it, customize the Am_TOP, Am_LEFT, Am_WIDTH, and Am_HEIGHT slots of the group to define its size, set the Am_INNER_WIDTH and Am_INNER_HEIGHT based on the contents, and add graphical parts to the group.

6.2.4.4.4 Simple Example

Here is a simple example of using a scrolling group.

```
my_scrolling_group = Am_Scrolling_Group.Create("scroll_group")
    .Set(Am_LEFT, 10)
    .Set(Am_TOP, 10)
    .Set(Am_WIDTH, 200)
    .Set(Am_HEIGHT, 300))
    .Add_Part(Am_Rectangle.Create()
        .Set(Am_LEFT, 0)
        .Set(Am_TOP, 0)
        .Set(Am_WIDTH, 15)
        .Set(Am_HEIGHT, 15)
        .Set(Am_FILL_STYLE, Am_Blue)
        );
my_window.Add_Part(my_scrolling_group);
```

This creates a scrolling group with an area of 200 by 300 pixels, and an internal region 400 by 400 pixels (the default values are inherited since none were specified). The scrolling group is displayed at location 10,10 in my_window. It contains a single object, a blue square 15 pixels on a side, in the upper left corner of the inner region. The scrolling group will have a vertical scroll bar on the right side of the group, and a horizontal scroll bar on the bottom of the group, as specified by the defaults.

6.2.5 Am_Text_Input_Widget

The Am_Text_Input_Widget is used to get a single line of text input from the user, for example for filenames.

Slot	Default Value	Type
Am_VALUE	""	Am_String
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	150	int
Am_HEIGHT	<formula>	int
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look
Am_FONT	Am_Default_Font	Am_Font
Am_LABEL_FONT	bold_font	Am_Font
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style
Am_ACTIVE_2	true	bool
Am_COMMAND	Am_Command	Am_Command

The widget has an optional label to the left of a text type-in field. The label is the value of the Am_LABEL field of the command object (and can be a string or arbitrary Amulet graphical object, or an empty string for no label). The user can click the mouse button in the field, and then type in a new value. The Am_VALUE of the command in the Am_COMMAND slot is set to the new string, and the command's Am_DO_METHOD is called. The command's default Am_UNDO_METHOD restores the Am_VALUE and the displayed string to its previous value. As the user types, if the string gets too long to be displayed, it scrolls left and right as appropriate so the cursor is always visible. The user can finish the typing by typing return on the keyboard, or by clicking outside the text input field.

The text input widget could also be used as an output-only text display by setting the Am_ACTIVE_2 slot (see Section 6.2.1) to false, which will disable the interactors.

The special slots of the Am_Text_Input_Widget are:

- Am_WIDTH - unlike most widgets, the default width is a constant (150) since the widget scrolls the text to fit. You will probably want to set the width to some other constant or formula.
- Am_HEIGHT - the default formula for the height uses the maximum of the height of the label and the height of the string.
- Am_FONT - this slot holds the font of the string that the user edits, and the default is the regular default font.

- Am_LABEL_FONT - this slot holds the font used for the label if the label is a string (the label comes from the Am_LABEL slot of the command object in the widget). The default is a bold font.
- Am_FILL_STYLE - the color used for the user type-in field.
- Am_WANT_PENDING_DELETE - if true (the default) then when the widget is explictly started (using Am_Start_Widget -- Section 6.5), the entire string is selected so the next character (unless it is a cursor movement character) will delete the entire string. Also, Am_WANT_PENDING_DELETE enables a double-click of the mouse to select the entire string so that the next character will delete the entire string.

6.2.5.1 Command in the Text Input Widget

The Am_LABEL of the command in the Am_Text_Input_Widget is used as the label of the input field, so if you do not want a label, make the slot be the null string "". The Am_VALUE of the widget is set with the value the user types, and the Am_DO_METHOD is called.

6.2.5.2 Tabbing from Widget to Widget

A built-in interactor, Am_Tab_To_Next_Widget_Interactor, and its command object support tabbing from one Am_Text_Input_Widget to another in the same window.

Slot	Default Value	Type	
Am_START_WHEN	Am_Input_Char ("ANY_TAB")	Am_Input_Char	*how starts*
Am_LIST_OF_TEXT_ WIDGETS	<formula>	Am_Value_List of Am_Text_Input_ Widgets	*default is all parts of owner of Interactor that are text widgets*
Am_PRIORITY	105	float or int	*should be larger than running priority (100)*

If you add an instance of the Am_Tab_To_Next_Widget_Interactor to a window or group, then by default, hitting the TAB key will move from the one Am_Text_Input_Widget to the next, and SHIFT_TAB will move backwards. As the cursor is moved to the Am_Text_Input_Widget, a box is drawn around the widget to show it is selected, and the widget is started with the old contents of the widget selected in "pending-delete" mode, so that if the next character typed is a normal printing character, it will replace the old string. If another widget was operating when the TAB key was hit, then that widget is stopped, which will call its DO_METHOD. There is no way to distinguish leaving a Am_Text_Input_Widget because the user hit RETURN to confirm the value, versus clicking outside, versus hitting TAB to go to the next field. Therefore, it is recommended that you do not use the DO_METHOD of the command in the widget to anything if you are using the Am_Tab_To_Next_Widget_Interactor. The command used by this interactor is not undoable, so moving from field to field is not queued on the undo history.

Note: In toolkits such as Motif and MS Windows, you can TAB to other widgets besides text input widgets, for example to use the arrow keys to select which button to press. This is not yet supported in Amulet, so the TAB interactor just goes from text widget to text widget, skipping all other kinds of widgets.

The Am_Tab_To_Next_Widget_Interactor has a number of slots that can be used to customize the behavior:

- Am_START_WHEN: Default value is Am_Inter_Char("ANY_TAB"), which accepts the TAB key with any modifier. The internal code checks for the SHIFT modifier down to decide whether to go forwards or backwards, but you can use any character to start this interactor.
- Am_LIST_OF_TEXT_WIDGETS: This must contain a list of Am_Text_Input_Widgets. By default, it contains a formula which computes the list of text widgets by looking through the owner of the interactor for all of its parts which are instances of Am_Text_Input_Widget. The default order of the widgets is back to front, so if you add your Am_Text_Input_Widgets to the group from top to bottom, left to right, this will be the correct order. If you want to compute the list of Am_Text_Input_Widgets some other way, simply set the Am_LIST_OF_TEXT_WIDGETS slot with the list sorted in the correct way, or else set it with a formula which returns the desired list.

6.2.6 Am_Selection_Widget

Am_Selection_Widget is used for selecting, moving, and resizing graphical objects.

Slot	Default Value	Type	
Am_VALUE	NULL	Am_Value_List	*list of selected objects*
Am_START_WHEN	Am_Input_Char ("ANY_LEFT_DOWN")	Am_Input_Char	
Am_FILL_STYLE	Am_Black	Am_Style	*color of handles*
Am_VALUE	Am_Value_List()	Am_Value_List	
Am_ACTIVE	true	bool	
Am_OPERATES_ON	NULL	Am_Object	*group to select from*
Am_START_WHERE_TEST	Am_Inter_In_Part	Am_Where_Method	
Am_COMMAND	Am_Command	Am_Command	*object selected*
Am_MOVE_GROW_COMMAND	Am_Command	Am_Command	*object moved/ grown*

Most graphical applications need to have "selection handles," which are small squares that show which object(s) are selected and which allow the objects to be moved and changed size. Surprisingly, most toolkits require each application to reimplement this basic functionality. Amulet supplies this behavior through the supplied Am_Selection_Widget object, which you simply can add to your application, and then its objects will be selectable and manipulatable

Slots that control the Am_Selection_Widget are:

- Am_OPERATES_ON: Must be set with a group, scrolling group or window that contains the objects that can be selected. The Am_Selection_Widget will allow the user to select any element of this group.

- Am_FILL_STYLE: The color of the selection handles. The default is Am_Black. The handles are drawn with this color on the inside, and a line style of Am_White. This insures that the handles are almost always visible, independent of the color of the objects and the background (XOR is not used, as in most other toolkits, since this often results in invisible handles).
- Am_ACTIVE: Controls whether the widget is active. Default is true.
- Am_START_WHEN: This is the character that the widget will start on. The default is Am_Input_Char("ANY_LEFT_DOWN").
- Am_START_WHERE_TEST: How to test the group in Am_OPERATES_ON for which objects to select. The default is Am_Inter_In_Part.
- Am_VALUE: The set of selected objects is supplied as an Am_Value_List in this slot. You can also set this slot with a Am_Value_List of objects you want to have selected.

6.2.6.1 Application Interface for Am_Selection_Widget

The Am_Selection_Widget you create should be added to a window or group in which you want the selection handles to appear. In general, it is a good idea to make the Am_OPERATES_ON group be a *different* group from the group that the Am_Selection_Widget is in. Typically, there will be a top level group, and the Am_OPERATES_ON group and the Am_Selection_Widget will be put into the top level group. For example:

```
//scroller is the top-level scrolling group to put things in
scroller = Am_Scrolling_Group.Create("scroller")
   .Set (Am_LEFT, 55)
   .Set (Am_TOP, 40)
   .Set (Am_INNER_WIDTH, 1000)
   .Set (Am_INNER_HEIGHT, 1000)
   .Set (Am_INNER_FILL_STYLE, Am_White)
   .Set (Am_WIDTH, scroll_width_formula)    //width and height will be
   .Set (Am_HEIGHT, scroll_height_formula) //  based on window's
   ;
//all objects that will be created and that can be selected and moved will be put into created_objs
created_objs = Am_Group.Create("created_objs")
   .Set (Am_LEFT, 0)
   .Set (Am_TOP, 0)
   .Set (Am_WIDTH, 1000)
   .Set (Am_HEIGHT, 1000)
   ;
//the selection widget operates on the group created_objs
my_selection = Am_Selection_Widget.Create("my_selection")
   .Set(Am_OPERATES_ON, created_objs)
//put the scroller in the window
my_window.Add_Part(scroller);
//put the selection widget and the created_objs as parts of the scrolling group
 scroller.Add_Part(created_objs);
scroller.Add_Part(my_selection);
```

As mentioned above, you can access the list of selected objects in the Am_VALUE slot of the Am_Selection_Widget. You can also set this slot to change the set of selected objects. Be sure to only set this slot to an Am_Value_List, even if you want no objects or a single object selected. For example, to clear the selection, use:

```
my_selection.Set(Am_VALUE, Am_Value_List());
```

There are also two command objects you can use to monitor the selection widget's activities. The Am_COMMAND part is used when the selection changes. The **Am_VALUE** of the Am_COMMAND object is set with the current selection, and its Am_DO_METHOD is called whenever the selection changes. The Am_IMPLEMENTATION_PARENT of this command is Am_NOT_USUALLY_UNDONE by default, since normally changing the selection is not undoable. If you want the selections to be undoable, set the Am_IMPLEMENTATION_PARENT slot of the Am_COMMAND of the selection widget to be NULL. This is done automatically by the undo dialog box (see Section 5.6.2.3.2) when the user clicks on the undo selections check box.

The other command is used when the user moves or grows an object. This command is in the Am_MOVE_GROW_COMMAND named part. You should probably *not* replace this command, because it has a built-in formula to make the label be correct based on the operation, but it is fine to override the various DO and UNDO methods. By default, the move and grow operations *are* undoable, if there is an undo handler attached to the window the selection widgets are in.

6.2.6.2 User Interface to Am_Selection_Widget

The selection widget operates in the standard way of the selection handles on the Macintosh and Windows. The user can click with the specified button (usually the left button) over an object to select it. Holding down the shift key while clicking will add or remove the object under the mouse to the selected set. Thus, to select multiple objects, you can click on them with the shift key held down. If you click in the background, all objects will be de-selected. Unfortunately, selecting objects by dragging out a region is not yet supported.

If you press down on an object and move the mouse, the object will be moved. If multiple objects are selected, they all will be moved. If you click on a selection handle, the object attached to the handle will be changed size. Note that it is currently not possible to grow multiple objects at the same time; only the object that the specific handle is attached to is grown.

6.3 Dialog boxes

Amulet provides three standard dialog box widgets with different appearances but similar operation to be used for simple messages and queries. We also provide several functions to make the dialog boxes easier to use. The dialog boxes are:

- Am_Alert_Dialog displays several lines of informational text, and an "OK" button below the text. It is used to alert the user of an error or to give the user information, and returns no value.
- Am_Choice_Dialog displays several lines of prompt text, and "OK" and "Cancel" buttons below the text. It is used to prompt the user to make a choice, and returns the value of the button they chose (either "OK" or "Cancel").

- `Am_Text_Input_Dialog` displays several lines of prompt text, a text input widget, and "OK" and "Cancel" buttons. It is used to prompt the user for text input. It returns the value of the text input widget if the user clicks on "Okay" or presses the RETURN key, and Am_No_Value if the user cancels.

6.3.1 Support functions for Dialog Boxes

Several functions are available to make it easier to use dialog boxes in common situations.

```
void Am_Show_Alert_Dialog (Am_Value_List alert_texts, int x = 100,
                    int y = 100, bool modal = false)
```

This routine brings up an alert dialog box at the location (x, y) on the main screen, and waits for the user to close it by clicking the "OK" button. The list `alert_texts` is a list of `char*` or `Am_String` values which will be displayed, one per line, above the "Okay" button. If `modal` is true, the dialog box will be run modally, otherwise it will be run non-modally. This routine does not return until the user clicks on either OK or Cancel.

```
Am_Value Am_Get_Choice_From_Dialog (Am_Value_List prompt_texts,
         int x = 100, int y = 100, bool modal = false);
```

This routine brings up a choice dialog box at the location (x, y) on the main screen, and waits for the user to close it by clicking either the "OK" or "Cancel" button. The list `prompt_texts` is a list of `char*` or `Am_String` values which will be displayed, one per line, above the buttons. If `modal` is true, the dialog box will be run modally, otherwise it will be run non-modally. The return value will be a string, either "OK" or "Cancel" depending on which button the user pressed.

```
Am_Value Am_Get_Input_From_Dialog (Am_Value_List prompt_texts,
                        Am_String initial_value = "",
                        int x = 100, int y = 100,
                        bool modal = false)
```

This routine brings up an input dialog box at the location (x, y) on the main screen, and waits for the user to close it by clicking either the "OK" or "Cancel" button. The list `prompt_texts` is a list of `char*` or `Am_String` values which will be displayed, one per line, above the text input line and buttons. If `modal` is true, the dialog box will be run modally, otherwise it will be run non-modally. The return value will be a string, the value of the text widget, if the user clicks "OK" or presses RETURN to close the dialog box. The routine returns `Am_No_Value` if the user clicks "Cancel."

```
Am_Value Am_Show_Dialog_And_Wait (Am_Object the_dialog,
                        bool modal = false)
```

This routine displays a preconfigured dialog box, waits for the user to complete interaction with it, and then returns its value. The only slots the dialog box changes before displaying the dialog box are its command's Am_DO_METHOD and Am_ABORT_METHOD. This routine is useful for reusing a commonly displayed dialog box without the overhead of reallocating and reinitializing the dialog box each time it is displayed. The dialog is added to Am_Screen if it is not already there, but it is not removed when the routine is finished.

6.3.2 Slots of dialog boxes

- Am_X_OFFSET: 5
- Am_Y_OFFSET: 5, The elements of the dialog box are put in a group inside the dialog box window. The Am_X_OFFSET and Am_Y_OFFSET are the size of the empty border area between this group and the edge of the window.
- Am_WIDTH: formula depending on contents.
- Am_HEIGHT: formula depending on contents. Dialog boxes automatically resize themselves to fit the dialog box elements with a border around them. You may override the formula by explicitly setting the Am_WIDTH and Am_HEIGHT slots, but then the dialog box will no longer resize itself.
- Am_FILL_STYLE: Am_Motif_Gray. This specifies the fill style of the dialog box window and all widgets contained in it.
- Am_WIDGET_LOOK: Am_MOTIF_LOOK
- Am_ITEMS: 0. Must be NULL or an Am_Value_List of char* or Am_String. This is a list of string items which will be displayed, one per line, above the dialog box widgets.
- Am_V_SPACING: 5. These slots control the layout of the multiple strings specified in Am_ITEMS
- Am_H_SPACING: 10
- Am_H_ALIGN: Am_CENTER_ALIGN
- Am_WIDGET_START_METHOD: Am_Show_Dialog_Method
- Am_WIDGET_STOP_METHOD: Am_Finish_Dialog_Method
- Am_WIDGET_ABORT_METHOD: Am_Finish_Dialog_Method
- Am_VALUE: "" or a formula depending on the user's selection. This contains the current value of a dialog box for choice and input dialog boxes. In input dialog boxes, it can be set with an initial value to be displayed in the text input widget.
- Other slots are inherited from Am_WINDOW.

6.3.3 Am_Text_Input_Dialog slots

- Am_VALID_INPUT: true. In text input dialog boxes, this slot tells the dialog box whether the text input field is a valid input or not. If Am_VALID_INPUT is false, the "OK" button will be inactive (greyed out), and the RETURN key will not close the dialog box (making "Cancel" the only valid operation). The intended use of this slot is to contain a constraint on the dialog's Am_VALUE slot which determines whether the input value is valid or not, and returns true or false appropriately.

6.4 Supplied Command Objects

Many operations should work the same way across many different applications. In particular, graphical editing commands such as cut, copy and paste should always work in a standard fashion. To help with this, Amulet supplies a set of pre-built Command objects that you can simply add to your menus or buttons to perform standard operations. These commands come complete with the complete UNDO methods, enabling methods, labels and accelerators, but you can of course override any of these as desired.

Most of these commands operate on the currently selected objects, so they require that you pass in an instance of a selection widget (Section 6.2.6) in the `Am_SELECTION_WIDGET` slot of the command object. If you implement your own selection handles and do not use the selection widget, you still may be able to use the following commands, if your selection handles object provides the set of selected objects in the `Am_VALUE` slot, and allows that slot to be set to change the selection.

The supplied command objects are as follows. See the tables in the Summary chapter, Section 10.7 for a list of the slots to be set in each.

- `Am_Selection_Widget_Select_All_Command`: used with the selection widget to cause everything to be selected. The label is "Select All" and the accelerator is "CONTROL_a". Must be passed a `Am_SELECTION_WIDGET`.
- `Am_Graphics_Clear_Command`: delete the selected objects. The label is "Clear" and the accelerator is "DELETE". Must be passed a `Am_SELECTION_WIDGET`.
- `Am_Graphics_Clear_All_Command`: The label is "Select All" and the accelerator is "CONTROL_a". Must be passed a `Am_SELECTION_WIDGET`.
- `Am_Graphics_Copy_Command`: Copy to the clipboard. The label is "Copy" and the accelerator is "CONTROL_c". Must be passed a `Am_SELECTION_WIDGET`.
- `Am_Graphics_Cut_Command`: Copy objects to the clipboard and then delete them. The label is "Cut" and the accelerator is "CONTROL_x". Must be passed a `Am_SELECTION_WIDGET`.
- `Am_Graphics_Paste_Command`: Paste a copy of the objects in the clipboard. The label is "Paste" and the accelerator is "CONTROL_v". Must be passed a `Am_SELECTION_WIDGET`.
- `Am_Graphics_To_Bottom_Command`: Make the selected objects be covered by all other objects. The label is "To Bottom" and the accelerator is "CONTROL_<". Must be passed a `Am_SELECTION_WIDGET`.
- `Am_Graphics_To_Top_Command`: Make the selected objects be covered by no other objects. The label is "To Top" and the accelerator is "CONTROL_>". Must be passed a `Am_SELECTION_WIDGET`.
- `Am_Graphics_Duplicate_Command`: Duplicate the selected objects. The label is "Duplicate" and the accelerator is "CONTROL_d". Must be passed a `Am_SELECTION_WIDGET`.
- `Am_Graphics_Group_Command`: Make a group out of the selected objects. Creates an instance of `Am_Resize_Parts_Group` (see Section 4.7.3 in the Opal chapter) and adds the selected objects to it. The new group has its `Am_CREATED_GROUP` slot set to `true`, which is used by

ungroup and change property. The label of the command is "Group" and the accelerator is "CONTROL_p". (We would have made it ^G, but that is used for aborting all interactors and widgets). Must be passed a Am_SELECTION_WIDGET.

- Am_Graphics_Ungroup_Command: Ungroup the selected objects. This will only operate on objects that have the Am_CREATED_GROUP slot set to true, which are typically those groups created by the Am_Graphics_Group_Command. The label is "Select All" and the accelerator is "CONTROL_h". Must be passed a Am_SELECTION_WIDGET.
- Am_Undo_Command: Perform a single undo. The label is "Undo" and the accelerator is "CONTROL_z". This finds the undo handler by looking for the window of the object in the Am_SELECTION_WIDGET slot (which can be any object whose Am_WINDOW is the window containing the undo handler--for this command, this slot doesn't have to contain a selection widget). If the Am_SELECTION_WIDGET slot is null, then gets the window from the widget the command is attached to, and looks for that window's undo handler.
- Am_Redo_Command: Perform a single redo. The label is "Redo" and the accelerator is "CONTROL_SHIFT_Z". Looks for an undo handler the same way as Am_Undo_Command.
- Am_Quit_No_Ask_Command: Quit the application immediately without asking for confirmation. The label is "Quit" and the accelerator is "CONTROL_q". Does *not* use an the Am_SELECTION_WIDGET.
- Am_Graphics_Set_Property_Command: to set the color, line style or other property of the selected objects. This has a slightly more complicated interface, as described in Section 6.4.2. Must be passed a Am_SELECTION_WIDGET.

As an example of the use of many of these commands, here is part of the menu bar definition for testselectionwidgets:

```
menu_bar = Am_Menu_Bar.Create("menu_bar")
  .Set(Am_ITEMS, Am_Value_List ()
   .Add (Am_Command.Create("File_Command")
         .Set(Am_IMPLEMENTATION_PARENT, true) //top command not queued for undo
         .Set(Am_LABEL, "File")
         .Set(Am_ITEMS, Am_Value_List ()
                    .Add (Am_Quit_No_Ask_Command.Create())
             )
         )
   .Add (Am_Command.Create("Edit_Command")
         .Set(Am_LABEL, "Edit")
         .Set(Am_IMPLEMENTATION_PARENT, true) //top command not queued for undo
         .Set(Am_ITEMS, Am_Value_List ()
               .Add (Am_Undo_Command.Create()) //these get the undo_handler from
               .Add (Am_Redo_Command.Create()) //     menubar's window
               .Add (Am_Show_Undo_Dialog_Box_Command.Create()
                   .Set(Am_UNDO_DIALOG_BOX, my_undo_dialog))
               .Add (Am_Menu_Line_Command.Create())
               .Add (Am_Graphics_Cut_Command.Create()
                   .Set(Am_SELECTION_WIDGET, my_selection))
               .Add (Am_Graphics_Copy_Command.Create()
                   .Set(Am_SELECTION_WIDGET, my_selection))
               .Add (Am_Graphics_Paste_Command.Create()
                   .Set(Am_SELECTION_WIDGET, my_selection))
               .Add (Am_Graphics_Clear_Command.Create()
                   .Set(Am_SELECTION_WIDGET, my_selection))
               .Add (Am_Graphics_Clear_All_Command.Create()
```

```
                    .Set(Am_SELECTION_WIDGET, my_selection))
                .Add (Am_Menu_Line_Command.Create())
                .Add (Am_Graphics_Duplicate_Command.Create()
                    .Set(Am_SELECTION_WIDGET, my_selection))
                .Add (Am_Selection_Widget_Select_All_Command.Create()
                    .Set(Am_SELECTION_WIDGET, my_selection))
                )
            )
        .Add (Am_Command.Create("Arrange_Command")
            .Set(Am_LABEL, "Arrange")
            .Set(Am_DO_METHOD, my_do)
            .Set(Am_IMPLEMENTATION_PARENT, true) //top command not queued for undo
            .Set(Am_ITEMS, Am_Value_List ()
                .Add (Am_Graphics_To_Top_Command.Create()
                    .Set(Am_SELECTION_WIDGET, my_selection))
                .Add (Am_Graphics_To_Bottom_Command.Create()
                    .Set(Am_SELECTION_WIDGET, my_selection))
                .Add (Am_Menu_Line_Command.Create())
                .Add (Am_Graphics_Group_Command.Create()
                    .Set(Am_SELECTION_WIDGET, my_selection))
                .Add (Am_Graphics_Ungroup_Command.Create()
                    .Set(Am_SELECTION_WIDGET, my_selection))
                )
            )
        )
    ;
```

6.4.1 Graphics Clipboard

The cut, copy, and paste commands operate on a clipboard object, which can be an arbitrary object whose `Am_VALUE` slot contains an Am_Value_List of objects. You can specify a particular clipboard to use by passing a clipboard object in the `Am_CLIPBOARD` slot of the command object. If this is NULL, Amulet uses the global `Am_Global_Clipboard` object. **Note that Amulet does not yet interoperate with the standard Windows or Macintosh clipboards.** The "clipboard" is currently just local to the Amulet application.

6.4.2 Am_Graphics_Set_Property_Command

The `Am_Graphics_Set_Property_Command` is designed to set properties like fill color, line style and fonts of graphical objects from menus or palettes. It iterates through all the selected objects setting a specified property to the value gotten from a widget (such as a palette or menu). If a selected object has the `Am_CREATED_GROUP` slot set to `true` (as do all groups created by the `Am_Graphics_Group_Command`), then the `Am_Graphics_Set_Property_Command` will recursively change the property of all of its parts. Of course, the `Am_Graphics_Set_Property_Command` is fully undoable and repeatable.

Because the command does not necessarily know how to get the correct value out of the palette or menu or how to update and read the property from the graphical object, a number of methods can be overridden in the command to control how the command gets and sets the value from the palette and from the graphical object. The slots that control the Am_Graphics_Set_Property_Command are:

- Am_SLOT_FOR_VALUE: The slot of the object and widget that the default methods use to get the value of. The default value is the Am_FILL_STYLE slot.
- Am_SELECTION_WIDGET: As with other commands in this section, the Am_SELECTION_WIDGET slot should be set with the selection widget that contains the list of selected objects whose property should be changed.
- Am_GET_WIDGET_VALUE_METHOD: The method in this slot should be of type
```
Am_Get_Widget_Property_Value_Method:
  Am_Define_Method_Type(Am_Get_Widget_Property_Value_Method, void,
                        (Am_Object command_obj, Am_Value &new_value));
```
The method should return (by setting the new_value parameter) the value that the widget currently is providing as the new value of the property. The default method uses the object the command is attached to as the widget, and gets that widget's Am_VALUE. If the contents of the Am_VALUE is an object, then that object's Am_SLOT_FOR_VALUE slot is accessed to get the value. Thus, if the widget is a button panel of where each item is a rectangle having the correct color, then the default method will correctly return the color of the item. Alternatively, if the widget is a menu, where each command in the menu has an Am_ID containing the correct value to use, the default method will return the correct value.
- Am_GET_OBJECT_VALUE_METHOD: The method in this slot should be of type
```
Am_Get_Object_Property_Value_Method:
  Am_Define_Method_Type(Am_Get_Object_Property_Value_Method, void,
                        (Am_Object command_obj, Am_Object object,
                         Am_Value &old_value));
```
The method should return (by setting old_value) the current value of the property for object, which is used in case the command needs to be undone. The default method just gets the value of the Am_SLOT_FOR_VALUE slot of object.
- Am_SET_OBJECT_VALUE_METHOD: The method in this slot should be of type
```
Am_Set_Object_Property_Value_Method:
  Am_Define_Method_Type(Am_Set_Object_Property_Value_Method, void,
                        (Am_Object command_obj, Am_Object object,
                         Am_Value new_value));
```
The method should set object so its property now has value new_value. The default method just sets the Am_SLOT_FOR_VALUE slot of object to be new_value.
- Am_SAVED_OLD_OWNER: As in all other command objects, this slot is automatically set with the widget the command is attached to. (Do not set this slot.) This is assumed by the default methods to be the widget from which the value is to be fetched.

6.5 Starting, Stopping and Aborting Widgets

Normally, widgets start, stop and abort running in response to the user's input events, but sometimes it is convenient to explicitly start and stop a widget. The routines in this section are useful for this.

```
extern void Am_Start_Widget(Am_Object widget,
                  Am_Value initial_value = Am_No_Value);
```

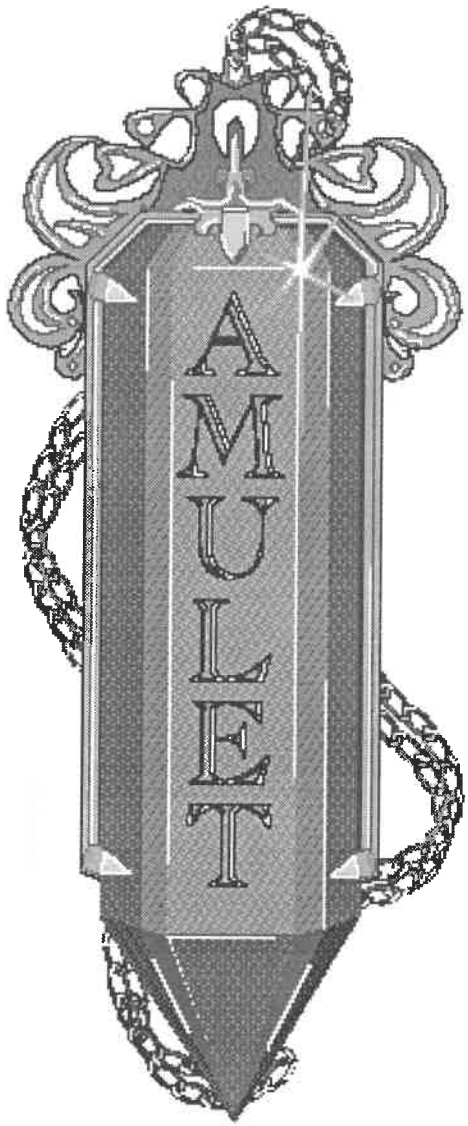
Explicitly start a widget running. If already running, does nothing. If an initial value is provided, then the widget is started with this as its value. It is up to the programmer to make sure the `initial_value` is legal for the type of widget. If no `initial_value` is supplied, the widget is started with its current value, if any.

```
extern void Am_Abort_Widget(Am_Object widget_or_inter_or_command);
```

Explicitly abort a widget, interactor or command object. Often, this will be called with a command object, and the system will then find the associated widget or interactor and abort it. If that widget or Interactor is not running, then this does nothing. The function tries to make sure the command object passed in (or the command object associated with the widget or Interactor) is not entered into the command history.

```
extern void Am_Stop_Widget(Am_Object widget,
                  Am_Value final_value = Am_No_Value);
```

Explicitly stop a widget. If not running, raises an error. If `final_value` is supplied, then this is the value used as the value of the widget. If `final_value` is not supplied, the widget uses its current value. Commands associated with the widget will be invoked just as if the widget had completed normally.



7. Gem: Amulet Low-Level Graphics Routines

This manual describes GEM, the low-level graphics system in Amulet. Gem provides a machine-independent layer so the rest of Amulet can work on different window managers without changing the code. Most programmers will not use the Gem layer, but it is available for advanced programmers who need especially efficient code.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

7.1 Introduction

Gem is the low-level graphics and input interface in Amulet that provides a machine independent API to all of the supported platforms' graphics and and event routines. Most Amulet programmers will not use the Gem layer, since Opal and Interactors provide all the functionality of the Gem layer, with the added convenience of automatic redrawing and functionality of the ORE object system. Opal, Interactors and the Amulet Widgets are written using Gem. We provide the Gem interface for advanced programmers who are concerned about performance, who want to extend Opal or create new widgets, and those who need the convenience of a machine independant graphical toolkit without the overhead of the Amulet object system.

Gem, which stands for the "Graphics and Events Manager", can be used independent of most of the rest of the Amulet. Gem uses the wrapper mechanism (for styles and fonts), but it does not use the ORE object system.

7.2 Include Files

The primary include file for Gem is `gem.h`. Gem also uses `types.h` for wrappers, `gdefs.h` for styles, images, point lists, and fonts, and `idefs.h` for the input events. For a complete description of Amulet include files and how to use them, see Section 1.6 in the Overview chapter.

7.3 Drawonables

The primary data structure in gem is the `Am_Drawonable`, which is a C++ object that corresponds to a window-manager window or off-screen buffer (for example, in X/11 it corresponds to a "drawable"). We called it a "Drawonable" because it is something that you can draw on. We also wanted to reserve the word "Window" for the Opal level object that corresponds to the drawonable. In this manual, sometimes "window" is used for "drawonable" since drawonables are implemented as window-manager windows.

7.3.1 Creating Drawonables

Programmers create a "root" drawonable at initialization, and then create other drawonables as children of the root (or as children of another drawonable). The typical initialization is:

```
Am_Drawonable *root = Am_Drawonable::Get_Root_Drawonable();
```

At the Opal level, this is called automatically by `Am_Initialize` to set up the exported `Am_Screen` object. Under X/11, `Get_Root_Drawonable` takes an optional string parameter which is the name of the screen. You can call therefore call `Get_Root_Drawonable` multiple times to support multiple screens.

Creating subsequent drawonables uses the `Create` method. If you use `root.Create` you get a top-level window, and if you use another drawonable, then it creates a sub-window. All of the parameters of the create call are optional, and are:

- `int l = 0`: the left of the new window in the coordinates of its parent.
- `int t = 0`: the top of the window
- `unsigned int w = 100`: width of the window
- `unsigned int h = 100`: height
- `const char* tit = ""`: the title for the window
- `const char* icon_tit = ""`: the string to display with the icon for the window.
- `bool vis = true`: whether the window is initially visible on the screen on not.
- `bool initially_iconified = false`: whether the window starts out as an icon.
- `Am_Style back_color = Am_No_Style`: the initial color for the background of the window.
- `unsigned int border_w = 2`: the size of the border of the window. This is ignored by most window managers for the top-level windows.
- `bool save_under_flag = false`: save the bitmaps underneath the window (useful for pop-up menus).
- `int min_w = 1`: The minimum size allowed for this window (when the user or program resizes it). You can't have 0 size windows.
- `int min_h = 1`: Minimum height.
- `int max_w = 0`: The maximum width allowed for the window. 0 is illegal so means no maximum.
- `int max_h = 0`: Maximum height.
- `bool title_bar_flag = true`: Whether the title line is displayed or not. Under X/11 having no title line means the window is not managed by the window manager.
- `bool query_user_for_position = false`: If true, then the initial left and top are ignored and the user is queried instead.
- `bool query_user_for_size = false`: If true, then the initial width and height are ignored and the user is queried instead.
- `bool clip_by_children_flag = true`: If false, then graphics drawn on the window show through all children windows (drawonables) created as children of this window.
- `Am_Input_Event_Handlers *evh = NULL) = 0`: How input is handled for this window, see Section 7.5.1.

To create an off-screen drawonable, you can use the shorter form:

```
virtual Am_Drawonable* Create_Offscreen (
  int width = 0, int height = 0,
  Am_Style background_color = Am_No_Style) = 0;
```

7.3.2 Modifying and Querying Drawonables

There are a number of methods on drawonables that query and set the various properties:

- `Am_Drawonable* Am_Drawonable::Narrow (Am_Ptr ptr)`: given an arbitrary pointer, this casts it to be a `Am_Drawonable`. Because drawonables are not wrappers, no checking is done.
- `void Destroy ()`: Destroys the drawonable and all its children (including removing them from the screen).
- `bool Inquire_Window_Borders(int& left_border, int& top_border, int& right_border, int& bottom_border, int& outer_left, int& outer_top)`: return the current window border sizes. For X/11, this may be inaccurate for windows that are not yet visible.
- `void Raise_Window (Am_Drawonable *target_d)`: Move the window to the "top" of all its siblings in Z order. If `target_d` is NULL, then the window is moved so it is not covered by any other windows. If `target_d` is a valid Am_Drawonable, then the window is put above `target_d` in Z order.
- `void Lower_Window (Am_Drawonable *target_d)`: Move the window to the "bottom" of all its siblings (if `target_d` is NULL), or just until it is below `target_d`.
- `void Set_Iconify (bool iconified)`: Make the window be iconified or not iconified.
- `void Set_Title (const char* new_title)`: Change window title.
- `void Set_Icon_Title (const char* new_title)`: Change icon title.
- `void Set_Position (int new_left, int new_top)`: Move the drawonable.
- `void Set_Size (unsigned int new_width, unsigned int new_height)`: Change the size.
- `void Set_Max_Size (unsigned int new_width, unsigned int new_height)`: Change the maximum size.
- `void Set_Min_Size (unsigned int new_width, unsigned int new_height)` : Change the minimum size.
- `void Set_Visible (bool vis)`: Make the window visible or not.
- `void Set_Border(bool new_title_bar, unsigned int new_width)`: Set whether has a title bar and how thick the border is.
- `void Set_Background_Color (Am_Style new_color)`
- `bool Get_Iconify ()`
- `const char* Get_Title ()`
- `const char* Get_Icon_Title ()`
- `void Get_Position (int& l, int& t)`: Sets l and t with current left and top.
- `void Get_Size (int& w, int& h)`
- `void Get_Max_Size (int& w, int& h)`
- `void Get_Min_Size (int& w, int& h)`
- `bool Get_Visible ()`
- `void Get_Border (bool& title_bar_flag, unsigned int& width)`

- `Am_Style& Get_Background_Color ()`
- `int Get_Depth ()`: Returns the current pixel depth in bits (e.g. 8 for 8-bit color).
- `bool Is_Color ()`: Returns true if the window is color or `false` if not.
- `void Get_Values (int& l, int& t, int& w, int& h, const char*& tit, const char*& icon_tit, bool& vis, bool& iconified_now, Am_Style& back_color, unsigned int& border_w, bool& save_under_flag, int& min_w, int& min_h, int& max_w, int& max_h, bool& title_bar_flag, bool& clip_by_children_flag, int& bit_depth)`: returns all of the parameters at once.

7.4 Drawing

Drawing in a drawonable is a three step process. First set the drawonable's clip region to the region you want to draw in. Then draw using the various drawing methods provided. Finally, you should "flush" the drawonable to process all pending drawing requests. The flush is necessary on X/11 to make the graphics appear. It is not strictly necessary on the PC or Macintosh, but to create machine independant code you should always explicitly call Flush_Output() on your drawonable if you want the output to appear.

The actual pixels drawn in a drawonable by the various drawing routines are described in detail in the Opal chapter of the Amulet manual. The Opal objects' slots are passed as parameters to the Gem level drawing routines. The `Am_Style`, `Am_Image_Array`, and `Am_Font` structures are also described there. That information will not be repeated here.

7.4.1 General drawonable operations

- `void Beep()`: Causes a sound on the machine this drawonable is displayed on. This is called by the Opal-level `Am_Beep()` routine.
- `void Set_Cursor(Am_Image_Array image, Am_Image_Array mask, Am_Style fg_color, Am_Style bg_color)`: Set the cursor for the drawonable. The mouse pointer will display the specified cursor whenever the mouse is within the bounds of this drawonable, and it will display the default cursor whenever the mouse exits a drawonable with a specific cursor. Amulet does not support setting a cursor globally, so that it has a custom image no matter where the mouse is pointing.
- `virtual void Bitblt (int d_left, int d_top, int width, int height, Am_Drawonable* source, int s_left, int s_top, Am_Draw_Function df = Am_DRAW_COPY)`: Bitblt copies a rectangular area from one drawonable to another, using the specified drawing function. The destination for bitblt is the `Am_Drawonable` this message is sent to.
- `void Clear_Area (int left, int top, int width, int height)`: This clears a rectangular region in the drawonable to the drawonable's backgound style.
- `void Flush_Output()`: This causes all pending output to be displayed on the screen. Under X/11, you will not see any graphics until this is called, unless you're running an event loop. Calling this routine is not strictly necessary on the PC or Mac, but you should call it occasionally to maintain machine independancy.

- `void Translate_Coordinates (int src_x, int src_y, Am_Drawonable *src_d, int& dest_x_return, int& dest_y_return)`: Convert the coordinates in one window to be coordinates in another window. To translate to or from screen coordinates, pass the root drawonable as the source or destination drawonable. Translating coordinates from a child of one root drawonable to a child of another root causes an `Am_Error()`. If the source or destination drawonable are offscreen drawonables, an `Am_Error()` will result.
- `void Translate_From_Virtual_Source (int src_x, int src_y, bool title_bar, int border_width, int& dest_x_return, int& dest_y_return)`: Translates a point from drawonable that hasn't neccessarily been created to screen coordinates. This function only works on root drawonables.

7.4.2 Size Calculation for Images and Text

In general, you cannot know the size of an image or a piece of text until you know where that image or text is going to be displayed. The same font may look different on different screens, depending on screen resolution, aspect ration, and so on. Therefore, the following are methods of drawonables rather than of images and font objects.

- `void Get_Image_Size (Am_Image_Array& image, int& ret_width, int& ret_height)`: Sets `ret_width` and `ret_height` with the width and height of the specified image array as if it were displayed on this drawonable.
- `int Get_Char_Width (Am_Font Am_font, char c)`: Returns the width in pixels of character `c` as if displayed in the font `Am_font` on this drawonable.
- `int Get_String_Width (Am_Font Am_font, const char* the_string, int the_string_length)`: Returns the width, in pixels, of the first `the_string_length` characters of `the_string`, as if it were displayed in font `Am_font` on this drawonable
- `void Get_String_Extents (Am_Font Am_font, const char* the_string, int the_string_length, int& width, int& ascent, int& descent, int& left_bearing, int& right_bearing)`: This returns the extents of the first `the_string_length` characters of `the_string`, as if drawn in font `Am_font` on this drawonable. The total height of the bounding rectangle for this string is `ascent + descent`. `left_bearing` is the distance from the origin of the text to the first "inked" pixel. The `right_bearing` is the distance from the origin of the text to the last "inked" pixel.
- `void Get_Font_Properties (Am_Font Am_font, int& max_char_width, int& min_char_width, int& max_char_ascent, int& max_char_descent)`: The `max ascent` and `max_descent` include vertical spacing between rows of text. The `max_char_width` and `min_char_width` are the width, in pixels, of the widest and narrowest characters in `Am_font` on this drawonable.

7.4.3 Clipping Operations

Gem supports clipping of all graphic operations to a specified clip region. Once a clip region is specified, all subsequent drawing operations are clipped so only parts inside the current clip region will show. To change the clip region of a drawonable, invoke one of the member functions listed below.

There is only one clip mask per root drawonable. All drawonables which have the same root share the same clip region. It is necessary to set the clip region for each window before drawing the contents of that window. For efficiency, it's a good idea to complete drawing in one window before resetting the clip region and drawing in other windows.

The clip region can be set with a rectangular region, or with an arbitrarily shaped Am_Region. Regions are described in Section 7.4.4. The root drawonable stores a stack of clipping regions. Use `Push_Clip()` and `Pop_Clip()` to push regions onto this stack, or to pop them off. The current clip region is the intersection of all of the regions currently on the stack.

- `void Clear_Clip()`: Clear the clip region and empty the clip region stack: no clipping will occur.
- `void Set_Clip (Am_Region* region)`: This empties the clip region stack and sets the clipping region to the specified `Am_Region`.
- `void Set_Clip (short left, short top, unsigned short width,unsigned short height)`: This empties the clip region stack and sets the clipping region to the rectangular region specified by `left`, `top`, `width`, and `height`.
- `void Push_Clip (Am_Region* region)`: This pushes the specified `Am_Region` onto the drawonable's clip region stack.
- `void Push_Clip (short left, short top, unsigned short width, unsigned short height)`: This pushes the rectangular region specified by `left`, `top`, `width`, and `height` onto the drawonable's clip region stack.
- `void Pop_Clip ()`: This pops the most recently pushed region off of the drawonable's clip region stack. Popping when there's nothing on the stack is silently ignored.

The `In_Clip()` routines provide a way to ask a drawonable if a point is inside its clip region. When asking if a given region is inside the drawonable's clip region, you can use the `total` parameter to determine whether the given region is completely inside the clip region, or whether it just intersects it.

- `bool In_Clip (short x, short y)`: Returns `true` if the point `(x,y)` is inside this drawonable's clip region, and `false` otherwise.
- `bool In_Clip (short left, short top, unsigned short width, unsigned short height, bool &total)`: Returns `true` if the rectangular region specified by `left`, `top`, `width`, and `height` intersects this drawonable's clip region. `total` is set to `true` if the clip region completely contains the rectangle, and `false` if part of the rectangle is outside the clip region.
- `bool In_Clip (Am_Region *rgn, bool &total)`: Returns `true` if `region` intersects this drawonable's clip region. `total` is set to `true` if the clip region completely contains `region`, and `false` if part of it is outside the clip region.

7.4.4 Regions

Instances of the `Am_Region` class describe arbitrarily shaped regions. `Am_Region` is a generalization of a drawonable's clip region, discussed in Section 7.4.3. By using the member functions listed below, you can build a region of arbitrary shape to install as the clip region of a drawonable.

```
class Am_Region {
public:

static  Am_Region* Create ();
  virtual void Destroy () = 0;
  virtual void Clear () = 0;
  virtual void Set (short left, short top, unsigned short width,
                    unsigned short height) = 0;
  virtual void Push (Am_Region* region) = 0;
  virtual void Push (short left, short top, unsigned short width,
                     unsigned short height) = 0;
  virtual void Pop () = 0;
  virtual void Union (short left, short top, unsigned short width,
                      unsigned short height) = 0;
  virtual void Intersect (short left, short top, unsigned short width,
                          unsigned short height) = 0;
  virtual bool In (short x, short y) = 0;
  virtual bool In (short left, short top, unsigned short width,
                   unsigned short height, bool &total) = 0;
  virtual bool In (Am_Region *rgn, bool &total) = 0;
};
```

7.4.5 Drawing Functions

All of the drawing functions take a `Am_Draw_Function` parameter which controls how the pixels of the drawn shape affect the screen. Since most programmers will use color screens, draw functions are not usually useful. The supported values for `Am_Draw_Function` are `Am_DRAW_COPY`, `Am_DRAW_OR` and `Am_DRAW_XOR`.

All of the following drawing routines draw in either an onscreen or offscreen drawonable. The parameters `ls` and `fs` specify the line style and fill style for the drawing operations.

- `void Draw_Arc (Am_Style ls, Am_Style fs, int left, int top, unsigned int width, unsigned int height, int angle1 = 0, int angle2 = 360, Am_Draw_Function f = Am_DRAW_COPY, Am_Arc_Style_Flag asf = Am_ARC_PIE_SLICE )`: Draw an optionally filled arc. To draw a circle, let `angle2` = 360.
- `void Draw_Image (int left, int top, int width, int height, Am_Image_Array image, int i_left = 0, int i_top = 0, Am_Style ls = Am_No_Style, Am_Style fs = Am_No_Style, Am_Draw_Function f = Am_DRAW_COPY)`: This draws the contents of an `Am_Image_Array` onto this drawonable. Bitmaps are treated differently than GIF pixmaps. With bitmaps, `ls` is used to control the color of 'on' bits, and `fs` is used for the background behind the image. If `fs` is `Am_No_Style`, the background pixels will be transparent- whatever was behind the image will show through. We do not support transparent GIF images on X/11 platforms. For GIFs and full color pixmaps, `fs` and `ls` are ignored.
- `void Get_Polygon_Bounding_Box (Am_Point_List pl, Am_Style ls, int& out_left, int& out_top int& width, int& height)`: Calculates the bounding box of the specified polygon.
- `void Draw_Line (Am_Style ls, int x1, int y1, int x2, int y2, Am_Draw_Function f = Am_DRAW_COPY)`: Draws a single straight line with style `ls`.

- `void Draw_Lines (Am_Style ls, Am_Style fs, Am_Point_List pl, Am_Draw_Function f = Am_DRAW_COPY)`: Draws an optionally filled polygon.
- `void Draw_2_Lines (Am_Style ls, Am_Style fs, int x1, int y1, int x2, int y2, int x3, int y3, Am_Draw_Function f = Am_DRAW_COPY)`: This is provided for more efficient drawing of an optionally filled polygon with exactly 2 lines.
- `void Draw_3_Lines (Am_Style ls, Am_Style fs, int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4, Am_Draw_Function f = Am_DRAW_COPY)`: This is provided for more efficient drawing of an optionally filled polygon with exactly 3 lines.
- `void Draw_Rectangle (Am_Style ls, Am_Style fs, int left, int top, int width, int height, Am_Draw_Function f = Am_DRAW_COPY )`: This draws an optionally filled rectangle.
- `void Draw_Roundtangle (Am_Style ls, Am_Style fs, int left, int top, int width, int height, unsigned short x_radius, unsigned short y_radius, Am_Draw_Function f = Am_DRAW_COPY)`: This draws a rectangle with rounded corners.
- `void Draw_Text (Am_Style ls, const char *s, int str_len, Am_Font Am_font, int left, int top, Am_Draw_Function f = Am_DRAW_COPY, Am_Style fs = Am_No_Style, bool invert = false)`: This draws a single line of text in the specified font. `ls` specifies the style of the text, and `fs` specifies the style of the rectangular region behind the text. If `fs` is `Am_No_Style`, the background is transparent.

7.5 Event Handling

Gem is usually used in event driven programs. In this style of program, there is a loop constantly running, checking for input events from the user, or other events the window manager might generate. These events are sent to Gem. When your program is ready to handle them, you tell Gem, and it dispatches one or more events to event handlers you have specified.

7.5.1 Am_Input_Event_Handlers

Gem dispatches events to the rest of your application by calling event handler routines you provide. The event handlers are stored as virtual methods of the C++ class `Am_Input_Event_Handlers`. You can specify one instance of this class for each of the drawonables in your application. The event handler methods take the drawonable and event information as parameters.

For example, each time the user hits a keyboard key over a certain window, that window's event handlers are retreived, and, the `Input_Event_Notify` member function is called, with information about what key was pressed, which drawonable received the keypress, and where the mouse was positioned within the window when the key was pressed.

Opal defines standard event handlers, so anyone programming at the Opal layer or above will not have to provide these event handlers. Instead, you should use an interactor for event handling, or create a new interactor if none of the available interactors is appropriate. To add or change the functionality of the standard opal event handlers, derive a new class from the C++ class `Am_Standard_Opal_Handlers` (exported in `opal_advanced.h`), and replace some of the virtual functions with your own event handlers.

`Am_Input_Event_Handlers` is defined as:

```
class Am_Input_Event_Handlers {
public:
  virtual void Iconify_Notify (Am_Drawonable* draw, bool iconified) = 0;
  virtual void Frame_Resize_Notify (Am_Drawonable* draw, int left,
                                    int top, int right, int bottom) = 0;
  virtual void Destroy_Notify (Am_Drawonable *draw) = 0;
  virtual void Configure_Notify (Am_Drawonable *draw, int left, int top,
                                 int width, int height) = 0;
  virtual void Exposure_Notify (Am_Drawonable *draw,
                                int left, int top,
                                int width, int height) = 0;
  virtual void Input_Event_Notify (Am_Drawonable *draw,
                                   Am_Input_Event *ev)=0;
};
```

The member functions are:

- `Iconify_Notify`: Called when the window is being iconified or de-iconified. The parameter `iconified` is true if the window is now iconified, and false if it was just de-iconified.
- `Frame_Resize_Notify`: Called whenever the window's border size changes (for example, if title bars are added or removed).
- `Destroy_Notify`: Called when the user requests that the window be destroyed. Note that window managers often don't actually destroy the windows, but rather call this routine to tell the programs to destroy the window.
- `Configure_Notify`: Called whenever the user changes the window's size or position.
- `Exposure_Notify`: Called when the window becomes uncovered and part of it needs to be redrawn. The rectangle specified by `left`, `top`, `width`, and `height` is the bounding box of the region that should be redrawn.
- `Input_Event_Notify`: Called for all input events from the keyboard and mouse. The `Am_Input_Event` is described below.

You can set and get the handlers in a particular drawonable using the following functions. If the event handlers are not set for a drawonable, they are inherited from the drawonable it was created from.

```
void Set_Input_Dispatch_Functions (Am_Input_Event_Handlers* evh)
void Get_Input_Dispatch_Functions (Am_Input_Event_Handlers*& evh)
```

7.5.2 Input Events

The input event passed to the `Input_Event_Notify` method is a C++ class containing the position of the mouse when the event occured, the drawonable of the event, a timestamp, and an `Am_Input_Char` describing the event. `Am_Input_Chars` are described in Chapter 5, *Interactors and Command Objects for Handling Input.*

```
class Am_Input_Event {
public:
    Am_Input_Char input_char; //the char and modifier bits; see idefs.h
    int x;
    int y;
    Am_Drawonable *draw; // Drawonable this event happened in
    unsigned long time_stamp;
};
```

You can control which input events are generated for a drawonable using the following member functions of drawonables:

- `void Set_Enter_Leave (bool want_enter_leave_events()`: If you want `Input_Event_Notify()` to be called when the mouse enters or leaves this drawonable, `Set_Enter_Leave(true)`. The default is `false`.
- `void Set_Want_Move (bool want_move_events)`: If you want a window to receive move events, set this to `true`. The default is `false`.
- `void Set_Multi_Window (bool want_multi_window)`: When an interactor should run over multiple windows, this method should be called on each window. Otherwise, the cursor is "reserved" for the original window the mouse is clicked in.
- `void Get_Window_Mask (bool& want_enter_leave_events, bool& want_move_events, bool& want_multi_window)`: This returns information about which input events a drawonable is receiving.

7.5.2.1 Multiple Click Events

On Unix and the Macintosh, we support multi-click events up to seven clicks. On the PC, we only support single and double clicks. You can control the time threshold interval between mouse downs for multiple click eventss. The exported global variable `Am_Double_Click_Time` is the inter-click wait time in milliseconds. The default value is 250. If it is 0, then no double-click processing is done. On the Mac, `Am_Double_Click_Time` is ignored. The global system click time is used instead. It is usually set from the Mouse Control Panel.

7.5.3 Main Loop

The normal Amulet program calls `Am_Initialize` (which among other things, calls `Get_Root_Drawonable`), then sets up a number of objects, and then calls `Am_Main_Event_Loop` or `Am_Do_Events` (Section 4.3.4). These routines then call a Gem level level routine where the events are actually processed. This routine is `Am_Drawonable::Process_Event()`. An Gem-level programmer who wants to process events, but *not* use any of the higher-level Amulet operations like demons and Opal might use `Am_Drawonable::Main_Loop`. This just repeatedly calls `Am_Drawonable::Process_Event ()`. To stop any of the main loops, you can set the exported `bool` called `Am_Main_Loop_Go` to false.

The difference between `Process_Event` and `Process_Immediate_Event` is that `Process_Event` waits for the next event, and processes exactly one input event and all non-input events (like refresh and configure_notify events) before and after that input event before returning. For example:

```
          before              after
       xxxIyyyIzzz    --->   Izzz
```

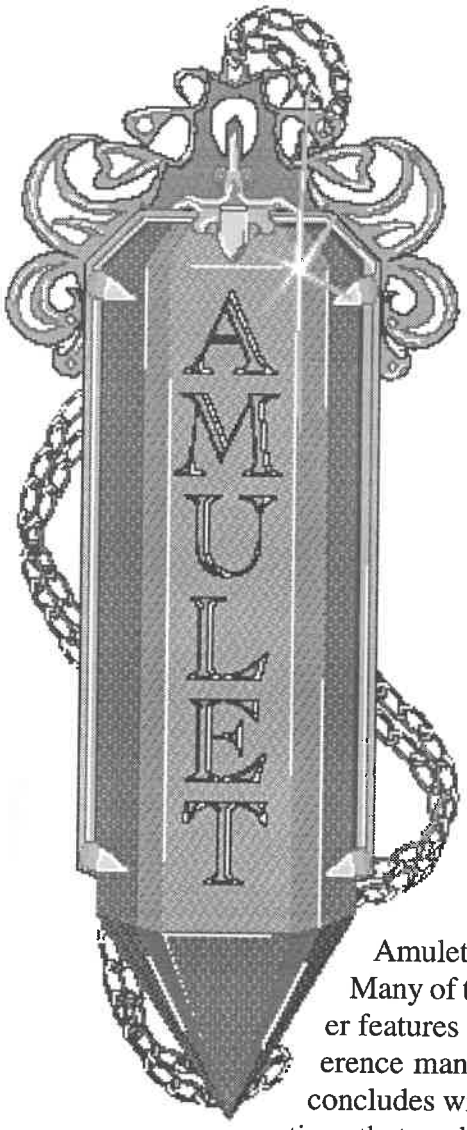
`Process_Event` returns when it encounters a second input event or when the queue is empty

`Process_Immediate_Event` does not wait for an event, but processes the first event in the queue and all non-input events after it until an input event is seen. `Process_Immediate_Event` returns when it encounters an input event (excluding the case where the first event is an input event) or when the queue is empty.

```
// Should Am_Drawonable::Main_Loop and Am_Main_Event_Loop keep running?
extern bool Am_Main_Loop_Go;

class Am_Drawonable {
public:
    ...
    static void Main_Loop ();
    static void Process_Event ();
    static void Process_Immediate_Event ();
    ...
```

`Am_Main_Event_Loop` uses `Process_Event`, and `Am_Do_Events` uses `Process_Immediate_Event`.



8. Debugging and the Inspector

Amulet contains many features to aide in debugging programs. Many of these are available interactively through the "Inspector". Other features are available programmatically. This chapter provides a reference manual for the Inspector and the other debugging facilities, and concludes with a set of recommendations about how to debug various situations that we have seen frequently.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

8.1 Introduction

We want to make Amulet programs very easy to develop and debug. Therefore, we have added extensive error checking to Amulet, as well as a number of interactive and tracing tools. These are designed to work with your regular C++ debugging tools like breakpoints. For example, the Inspector will allow you to break into the debugger when a slot is set, but then you need to use your regular C++ debugger to figure out why the slot was set. The debugging features are all implemented using machine-independent code, except for a single routine that breaks into the debugger. Therefore, we assume you can do stack traces and look at variables in your debugger, rather than needing to do this in our tools.

We would very much like to enhance the debugging capabilities of Amulet. If you think of a facility that would be useful, please let us know. An article about the debugging facilities in Amulet is available from the Amulet web site.

8.2 Include Files

The interactive Inspector is included by default in the Amulet library when you build the debugging option (see Section 8.4 of the Overview document). To use the Inspector, if you have the debugging library, you do not need to do anything special in your application. However, if you want to use any of the debugging features procedurally, you must include the `debugger.h` file, because it is *not* included by default when you include `amulet.h`. The most portable way to include the debugger header file is:

```
#include <am_inc.h>
#include DEBUGGER__H
```

8.3 Inspector

The Inspector is an interactive program that provides access to a large number of debugging features. At its most basic, it displays the slots of an object in a window. The values of most types of slots can be edited. The properties of the slots and the object can be inspected, as well as dependencies of any formulas in the slots. Traces and breakpoints can be set when slots are accessed or set. The Interactor's tracing mechanisms are also available from the Inspector.

8.3.1 Invoking the Inspector

To pop up the Inspector on an object, you can put the cursor over the object, and hit the F1 key (see Section 8.3.1.1 about how to change the keys, if necessary). The inspector will first print out the window and location in the window of the cursor to the transcript (console) window, and then a new top-level window will appear displaying all the slots of the object and their current values. If no object is under the cursor, then the window itself will be inspected. The Inspector first tries to find a primitve (leaf) object under the cursor, but if that fails, then it presents the front-most group object. If the correct object does not appear, it is usually best to get to the object by going up and down the owner/part tree, as will be explained below.

If you press the F2 key over an object, the Inspector will do the same search for an object, but will only print out the position in the window and the object at that position. This is also very useful for finding out the coordinates of a point in the window.

The F3 key will ask in the console window for the name of an object to inspect. The name typed must be the exact name of the object, which is sometimes an easy-to-remember constant name (like Amulet's or the user's prototypes) or often cut and pasted from the output of one of the tracing functions.

The inspector can also be invoked procedurally using either of the functions:

```
// inspect the specific object
void Am_Inspect(Am_Object object);

// The next one takes the name of the object. This is useful from an interpreter.
void Am_Inspect(const char * name);
```

8.3.1.1 Changing the Keys

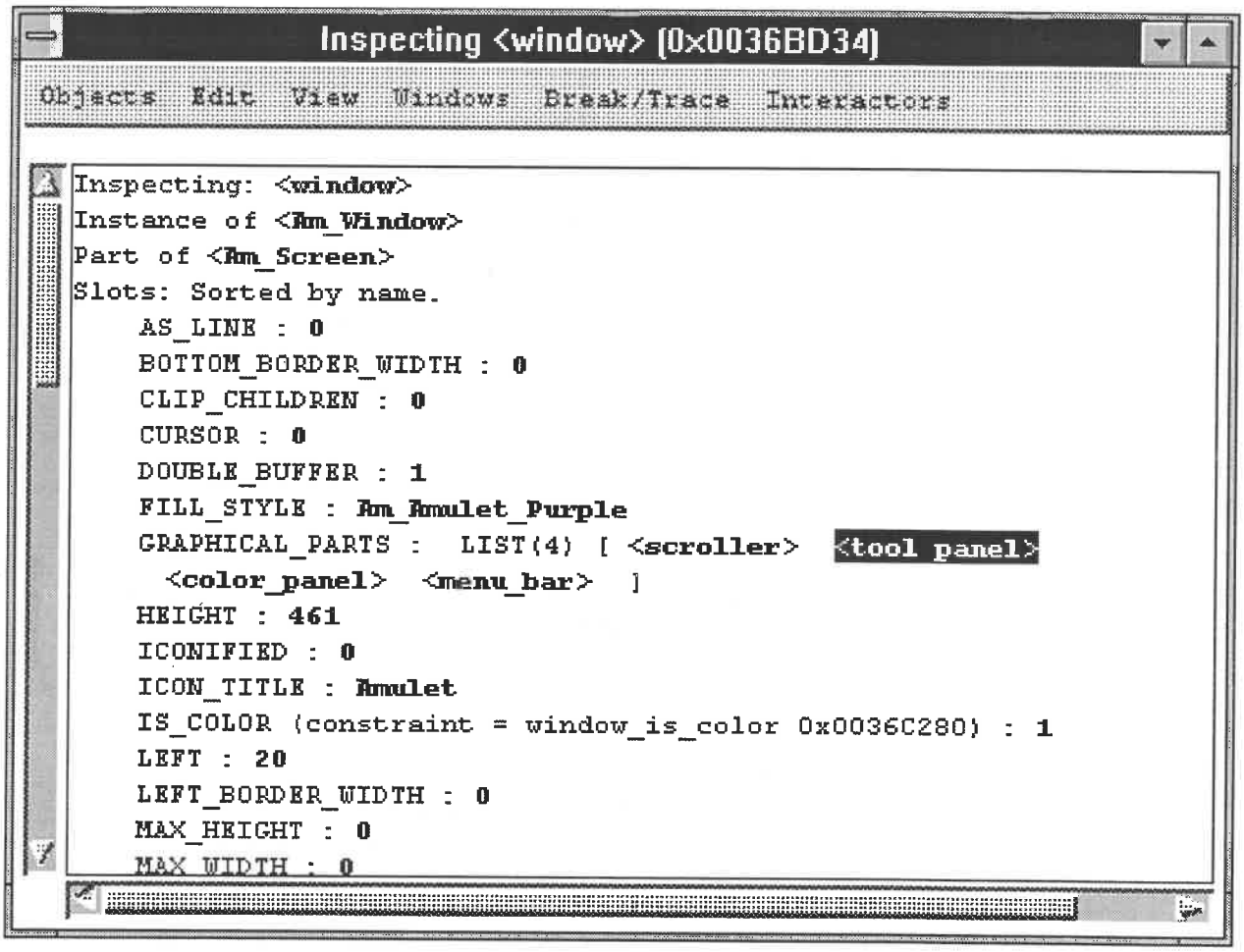
If you want to use F1, F2, and F3 for your own functions, you can always eliminate all of debugging facilities, including the inspector, by building a non-debug version of the library (see the Overview Chapter).

If you want to bind the editing functions to different keys, you can use the function:

```
extern void Am_Set_Inspector_Keys(Am_Input_Char show_key,
                                  Am_Input_Char show_position_key,
                                  Am_Input_Char ask_key);
```

which is exported from `debugger.h`. This function takes an Am_Input_Char, but a string will also work. Passing in the null character to any of these, by using `Am_Input_Char()`, will mean that that function is not available. The first parameter controls the normal popping up of the Inspector (normally bound to F1), the second parameter is for just printing the object under the mouse (normally F2), and the third is to prompt from the keyboard (normally F3). The test program testinter has a test of rebinding the Inspector's keys using this function.

8.3.2 Overview of Inspector User Interface and Menus



The inspector window is shown above. All of the slots of the objects are shown along with their current values. Inherited slots are shown in blue, and local slots are shown in black. You can click on a slot value to show a cursor, and then the value can be edited.

You can double-click on a slot name, an object name, or a constraint name to select it, and then perform other operations on that slot, object or constraint (such as viewing its properties). For the commands which pop-up windows, you can select names in those windows as well by double-clicking on them. The name which is selected by double-clicking is also copied into the cut buffer (clipboard) so it can be pasted into this or other applications.

Also shown in the Inspector window, below the list of slots (in this case, you would need to scroll down using the scroll bar on the left) is (optionally) a list of the *parts* of the object, and the *instances* of the object.

Commands available from the menus are (many of these are described in detail below):

- **Objects**: This menu contains commands that operate on objects, and also the "Done" commands (that would normally be in a "File" menu). Commands in this menu are:
  - **Inspect Object**: If an object is selected (by double-clicking with the left button on an object name), then this command inspects that object in this window. Accelerator for this is ^i (for "Inspect"), but you can also point at an object name with the right button, without even selecting the name first.
  - **Inspect in New Window**: If an object name is selected, then this will cause it to be inspected in a new window. Accelerators are ^-shift-I, or pointing at the object name using SHIFT-right-button.
  - **Inspect Object Named...**: This pops up a dialog box into which you can type an object name. If the name has been put into the window manager's clipboard (using cut or copy, or equivalent), then you can paste the name using the middle mouse button or ^y (for "yank"). The new object is inspected in the same window.
  - **Inspect Previous**: Shows the previous object shown in this window again. There is a stack of all the objects shown in each inspector window, and this effectively does a "pop". Accelerator is ^p.
  - **Refresh**: Most of the time, the inspector window will correctly refresh to show the current values of the object. Sometimes, however (especially if you turn off Automatic Refresh) the display will not correctly show the state of the object, in which case you should use the Refresh command to get the current values of all of the slots.
  - **Flash Object**: If an object is selected (by double-clicking on its name), then this will cause the object's window to come to the front, and the object to flash. If the object is not visible for some reason, then this prints an explanation of why not *to the console window* (eventually, we will print it to a dialog box). Thus, this command is useful for seeing where objects are, or why they might not be visible. If no object is selected, then this command operates on the current object being inspected. Accelerator is ^f.
  - **Done**: This gets rid of the current inspector window. Accelerator is ^q (for quit). You can also use the regular window manager mechanism for getting rid of windows, like "Kill Window" under Motif, or the "Close" command under Windows NT.
  - **Done All**: This gets rid of all inspector windows.
  - **Quit Application**: This kills the application and all the inspector windows. This does not ask for confirmation, and it does not notify the application or try to recover gracefully, it just exits the main event loop.
- **Edit**: This menu will eventually contain all the commands to edit the values of slots. For now, it only has one sub-command:
  - **Remove Slot**: If a slot name is selected (by double-clicking on the slot name), then this command will allow that slot to be removed. This can be used to cause a local slot to go back to being inherited.

- **View**: This menu contains various commands for specifying what is viewed in the main window, and how. Most of the sub-commands are toggles, and switch between two labels.
  - **Hide Inherited Slots / Show Inherited Slots**: By default the inherited and local slots are all shown. This command will toggle whether the inherited slots are shown at all.
  - **Hide Internal Slots / Show Internal Slots**: By default, the internal and normal slots of the object are all shown. By convention, the internal slots are those whose names begin with a tilde (~). There is no system-defined enforcement of internal vs. regular slots currently. It is generally a bad idea to change the values of internal slots. This toggles whether the internal slots are shown.
  - **Show Parts / Hide Parts**: By default, all of the parts of an object are shown below the list of the slot names (you may need to scroll down). This command toggles whether the parts are shown.
  - **Show Instances / Hide Instances**: By default, the instances of the object are *not* shown. This command will toggle whether all the instances of the object are shown below the list of parts at the bottom of the window (you may need to scroll down).
  - **Automatic Refresh / Manual Refresh**: By default, whenever a value in the inspected object changes, the display is refreshed to always show the current value (*note that this is not guaranteed*; it is best to use the Refresh command if you want to guarantee the values are up-to-date). However, updating the inspector may *substantially* slow down the execution of your program. Therefore, you might want to toggle to Manual Refresh, which means that the inspector window will not be updated until you explicitly call the Refresh command (in the Object menu).
  - **Show Old Slot Values**: This is an experimental feature that displays the old values of the slot, as well as the current value. It only works for some types of values, and there is currently no way to get rid of the display except to get rid of the inspector window. Let us know if you find this feature useful, and we may make it more robust.
  - **Sort by Name / Stop sorting by Name**: By default, the display of the slots is sorted by the slot name in alphabetical order. This toggle will cause the slots to instead be shown in the order they appear in the object, which is somewhat faster.
- **Windows**: This menu contains commands that pop up property windows that give more information about the selected object, slot or constraint. If you select a different object, slot or constraint, the pop-up window will move and show the properties of that one instead. You can get rid of these pop-up windows using the regular window manager mechanism (such as the close box or Kill Window window-manager command). Just like in the main Inspector window, you can double-click on objects, slots and constraints listed in these pop-up windows to select them (or right-button clicking for objects). However, you cannot change (edit) any values in the pop-up windows.
  - **Show Prototypes and Owners**: If an object name is selected, then this pops up a window that shows the object's name, its prototype's name, and that object's

prototype, etc. all the way up to the root object. Similarly, it also shows the owners all the way up to the screen (or to an object with no owner for objects that are not on the screen). If no object name is selected, then this operates on the object being inspected.

- **Show Constraint Dependencies**: If the name of a formula is selected, then this pops up a window that shows the slots that the formula is currently depending on, and the current values of those slots. If those slots also contain a formula, then the dependencies of those are shown also, down to a depth of three. You can always double-click on a constraint in this window, to see its dependencies.
- **Show Slot Properties**: If a slot name is selected, then this pops up a window that displays a number of properties of the slot. Most of these are the advanced properties such as declarations for inheritance and the demon bits. The most useful non-advanced properties are that for inherited slots, it shows the object from where the slot is being inherited from (which will be somewhere up the prototype chain), and the current type of the value in the slot.
- **Show Slot Uses**: If a slot name is selected, then pops up a window that shows all the formulas which use this slot. This will show you what will be affected if the slot value changes.

- **Break/Trace**: This menu contains commands that allow breaks and traces to be set on slot setting, either by direct setting or by constraints being re-evaluated. "Traces" print to the console, and "breaks" cause your C++ debugger to be entered (so you should be running the application in the debugger before using breaks).
  - **Break into C++ Debugger Now**: Breaks into the debugger.
  - **Trace When This Slot Set**: If a slot name is selected, then if the slot is set, then prints information about why it was set to the console.
  - **Break into C++ Debugger When This Slot Set**: If a slot is selected, then when the slot is set, the reason why is printed to the console, and then breaks into the C++ debugger so you can look at the stack trace to figure out what procedures or constraints are being executed.
- **Interactors**: This menu provides access to the various Interactor tracing facilities described in Section 5.7 of the Interactors chapter. All of these output to the console window.
  - **Turn off Interactor Tracing**: Cause there to be no tracing.
  - **Trace This Interactor**: If an object is selected and that object is an Interactor, then traces all actions by that interactor.
  - **Trace Interactor Named...**: Pops up a dialog box that allows the user to type in the name of an Interactor to be traced.
  - **Trace All Interactors**: Prints complete information for all interactors that run.
  - **Trace Next Interactor To Run**: Waits for the next interactor to run, and then starts complete tracing of only that interactor.
  - **Trace Input Events**: Only prints out information about the input events that happen in any window.
  - **Trace Interactor Set Slots**: Prints out information about any slot set by any

interactor. This is very useful if the problem is that a slot appears to be set erroneously.

- **Trace Interactor Priorities**: Prints out information about when interactors change priority levels. This is not particularly useful.
- **Short Trace Interactors**: Just prints out which interactors start, run, stop and abort.

8.3.3 Viewing and Editing Slot Values

When you inspect an object, the values of the slots are displayed. You can control which slots are viewed, by hiding or showing the inherited slots and/or the internal slots. You can also control the sorting of the slots (either alphabetical or in the order they appear in the object).

If you single click with the left mouse button over a slot value, a cursor will appear and you can edit the slot's value. The editing keys are the same as for all other text interactors (Section 5.3.5.5.1 of the Interactors chapter). Currently, the inspector will *not* let you change the *type* of the value in the slot. Therefore, it uses the current type of the value to decide how to parse the input value. You can edit primitive values, like integers, floats and strings. Depending on whether your compiler supports bools as a primitive type, they will either print out as true and false or 1 and 0. Floating point values print out just like integers if there is no fraction part. You can use the slot properties pop-up window to find out the exact slot type.

For slots which contain named values, like styles (`Am_Red`, `Am_Line_8`), objects (`Am_Rectangle_123`), constraint names (`windows_is_color`) and method names (`rectangle_draw`), you can type in a new name. Amulet remembers the names of all built-in or user-defined objects, methods and formulas. If you create your own styles or wrappers, you can arrange for them to have names registered in the database using the "registry" mechanism defined in registry.h. For any wrapper object, you can register its name using the `Am_Register_Name` procedure, such as:

```
Am_Register_Name (my_color_object, "my_color");
```

Then, the user will be able to type in `my_color` as the value of a slot.

Unfortunately, you cannot yet set the value of slots that are `Am_Value_Lists`, and you cannot set the items of an `Am_Value_List`.

8.4 Accessing Debugging Functions Procedurally

Sometimes it might be useful to access the debugging functions from a program, instead of interactively from the Inspector. For example, you program might be crashing even before it fully starts up, so you cannot access the inspector. If you program gets past the Am_Initialize(), then you can still trace slot setting and print the values of the slots of objects. Also, some of these procedures might be executed from a debugger such as gdb that supports calling functions.

The functions for invoking the Inspector procedurally have already been listed:

```
// inspect the specific object
void Am_Inspect(Am_Object object);
// The next one takes the name of the object. This is useful from an interpreter.
void Am_Inspect(const char * name);
```

You can cause an object to be "flashed" so you can see where it is on the screen. If it is not visible, then this functions writes the reason to the specified stream:

```
void Am_Flash (Am_Object o, ostream &flashout = cout);
```

The tracing functions provide significantly more features than are available interactively from the inspector. The tracing and breaking function takes an optional object, an optional slot, and an optional value. Whatever ones of these are supplied will control whether to trace or break. Thus, if only the object is supplied, then the trace or break will happen whenever any of the slots of that object are set. If only a value is supplied, then a trace or break will happen whenever any slot of any object is set to that value. If all three parameters are supplied, then a trace or break will happen only when that slot of that object is set to that value.

```
void Am_Notify_On_Slot_Set (Am_Object object = Am_No_Object,
                            Am_Slot_Key key = 0,
                            Am_Value value = Am_No_Value);
void Am_Break_On_Slot_Set (Am_Object object = Am_No_Object,
                           Am_Slot_Key key = 0,
                           Am_Value value = Am_No_Value);
```

The next procedure clears a slot notify or break set with the above procedures:

```
void Am_Clear_Slot_Notify (Am_Object object = Am_No_Object,
                           Am_Slot_Key key = 0,
                           Am_Value value = Am_No_Value);
```

8.5 Hints on Debugging

This section lists some hints of procedures we have found useful for debugging certain situations that we have found occur more than once. If you know anything that should be added to this list, please let us know!

1) **I click on an object but nothing happens**: If you think that an interactor should be running, but it doesn't, it is usually very helpful to turn on Interactor tracing from the Inspector, and see what interactors are running, and what slots they are setting. Although this often prints out a tremendous amount of information, and significantly slows down execution, it is usually worth it. Typical reasons that Interactor's don't seem to execute include:

   1.1) The Interactor was never added as a part to an object which is a part of a window (or a group in a window, recursively).

   1.2) The `Am_START_WHERE_TEST` of the Interactor does not return an object when you expect it to. Debugging this usually involves setting breakpoints in your start where function.

   1.3) The Interactor is running, but you can't see the feedback object, because the feedback object was never added to a window or group in a window.

   1.4) The Interactor is running, but the object it is setting slots of is not the object

you expected, so setting the slots is not having any effect.

2) **My Interactor is modifying the group itself instead of the members of the group, or affecting the members of a group, and I want it to affect the group itself:** The default `Am_START_WHERE_TEST` of Interactors tries to be smart about which object to affect, based on the type of the object the Interactor is attached to. You can specifically tell the Interactor which to affect by setting the `Am_START_WHERE_TEST` slot to one of the built-in functions (such as `Am_Inter_In` or `Am_Inter_In_Part`) or to a custom function.

3) **The slot has the wrong value:** There are various reasons this might happen:

   3.1) If the value is supposed to be calculated by a constraint, see if the constraint was accidentally removed (see number 8).

   3.2) Check if there are accidentally *two* or more constraints in the slot. Sometimes the system sets single-constraint-mode to false and installs an important constraint in the slot, which might override the value you set. To get rid of an inherited constraint, you need to delete the slot in the prototype object, or set the slot to single constraint mode and set its value.

   3.3) Maybe the code is setting the slot of the wrong object: put a breakpoint in your code to see what object is being set.

   3.4) Maybe your code is setting the slot, but then some other code, like an Interactor or command object's DO method, is setting it afterwards to a different value: put a break or trace on the slot using the Inspector to see when it is being set.

4) **I want to set a widget to have a certain value, but it isn't working:** To set the value of a widget, you have to set the `Am_VALUE` slot *of the widget itself*. Although the `Am_VALUE` slot of the command object usually provides the same value, it *does not* affect the widget to set this slot in the command object. For button-type widgets, the value should either be NULL, or the label or ID of the particular item to become the value (based on the value of the `Am_LABEL` slot or `Am_ID` slot of the command object for that item).

5) **I set the value of a button, or click on it, but I can't see the value:** By default, buttons do *not* stay "indented" when they are pressed. If you want a button to show the value, you have to set its `Am_FINAL_FEEDBACK_WANTED` slot to true.

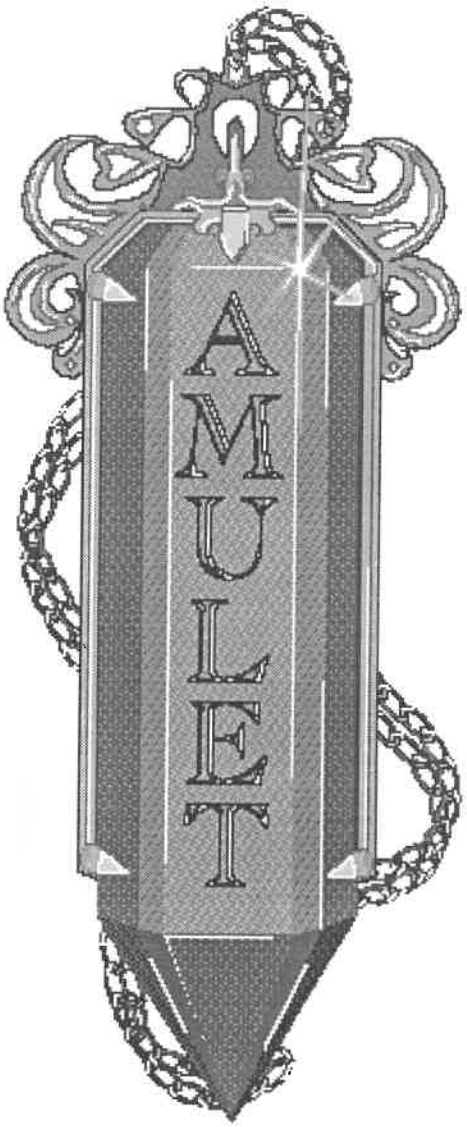
6) **My constraint is crashing or calculating the wrong value:** Because constraints are normal C++ code, you can used your debugger to set breakboints in the constraint code to see why it is calculating its current value. Also, in the Inspector, you can select the slot the constraint is in and set a trace or break on the slot to see when the slot is changing value.

7) **My application crashes at start up:** Usually this is caused by constraints calculating incorrect values. Amulet supports "Uninitialized" values for constraints, but sometimes this is not sufficient to allow constraints to work. You probably need to put a few `obj.Valid()` tests, or instead of getting a slot value directly into a variable, use an intermediate `Am_Value`. For example:

```
~~i = 3 / (int)obj.GV(SLOT);  //crashes because returns 0 when not initialized~~
Am_Value v;
obj.GVM(SLOT, v); //remember to change macro to be GVM
if (v.Valid()) {
   i = 3 / (int)v;
```

8) **The constraint in my slot disappeared:** Remember that constraints go away by default when the slot is set, so if an interactor or your program sets the slot, the constraint will be removed. You can see when the slot value is set by setting a break or trace on slot setting using the Inspector. To make sure the constraint is retained, you can set single-constraint-mode to be false for the slot. See Section 3.11.5 of the Ore chapter.
```
obj.Set_Single_Constraint_Mode(SLOT,false);
```

9) **The constraint is not being re-evaluated when it should be:** Usually, this is because there isn't a dependency where there should be. Make sure that the references in the formula use `GV` or `GVM` and not `Get` or else no dependency will be established. You can select the constraint in the Inspector and check its dependencies. Also, it might be useful to trace or break when the slot's value changes and when the depended on slot's value changes, to make sure that they are really changing.



10. Summary of Exported Objects and Slots

This chapter provides a summary of all the objects and slots exported by Amulet that the normal Amulet programmer will use. The specifics of the operation of the objects are discussed in other chapters of this manual.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

10.1 Am_Style

10.1.1 Constructors and Creators

```
Am_Style (float red, float green, float blue,          //color part
        short thickness = 0,
        Am_Line_Cap_Style_Flag cap_flag = Am_CAP_BUTT,
        Am_Join_Style_Flag join_flag = Am_JOIN_MITER,
        Am_Line_Solid_Flag line_flag = Am_LINE_SOLID,
        const char* dash_list = Am_DEFAULT_DASH_LIST,
        int dash_list_length = Am_DEFAULT_DASH_LIST_LENGTH,
        Am_Fill_Solid_Flag fill_flag = Am_FILL_SOLID,
        Am_Fill_Poly_Flag poly_flag = Am_FILL_POLY_EVEN_ODD,
        Am_Image_Array stipple = Am_No_Image)

Am_Style (const char* color_name,                      //color part
        short thickness = 0,
        Am_Line_Cap_Style_Flag cap_flag = Am_CAP_BUTT,
        Am_Join_Style_Flag join_flag = Am_JOIN_MITER,
        Am_Line_Solid_Flag line_flag = Am_LINE_SOLID,
        const char *dash_list = Am_DEFAULT_DASH_LIST,
        int dash_list_length = Am_DEFAULT_DASH_LIST_LENGTH,
        Am_Fill_Solid_Flag fill_flag = Am_FILL_SOLID,
        Am_Fill_Poly_Flag poly_flag = Am_FILL_POLY_EVEN_ODD,
        Am_Image_Array stipple = Am_No_Image)

static Am_Style Thick_Line (unsigned short thickness);
static Am_Style Halftone_Stipple (int percent,
        Am_Fill_Solid_Flag fill_flag = Am_FILL_STIPPLED);
```

10.1.2 Style Accessors

The following methods of `Am_Style` are available to query the properties of styles:

```
void Get_Values (float& red, float& green, float& blue) const;
void Get_Values (short& thickness,
        Am_Line_Cap_Style_Flag& cap, Am_Join_Style_Flag& join,
        Am_Line_Solid_Flag& line_flag,const char*& dash_l,
        int& dash_l_length, Am_Fill_Solid_Flag& fill_flag,
        Am_Fill_Poly_Flag& poly, Am_Image_Array& stipple) const;

void Get_Values (float& r, float& g, float& b, short& thickness,
        Am_Line_Cap_Style_Flag& cap, Am_Join_Style_Flag& join,
        Am_Line_Solid_Flag& line_flag, const char*& dash_l,
        int& dash_l_length,Am_Fill_Solid_Flag& fill_flag,
        Am_Fill_Poly_Flag& poly,Am_Image_Array& stipple) const;

Am_Fill_Solid_Flag Get_Fill_Flag() const;
Am_Image_Array Get_Stipple() const;
Am_Fill_Poly_Flag Get_Fill_Poly_Flag () const;

//Get the properties needed to calculate the line width
void Get_Line_Thickness_Values (short& thickness,
        Am_Line_Cap_Style_Flag& cap) const;

const char* Get_Color_Name () const; //returns a pointer to the string, don't dealloc
```

10.1.3 Color styles

The following color styles have line thickness zero (which really means 1, as explained in Section 4.6.3.2)

Am_Red	Am_Cyan	Am_Motif_Gray	Am_Motif_Light_Gray
Am_Green	Am_Orange	Am_Motif_Blue	Am_Motif_Light_Blue
Am_Blue	Am_Black	Am_Motif_Green	Am_Motif_Light_Green
Am_Yellow	Am_White	Am_Motif_Orange	Am_Motif_Light_Orange
Am_Purple		Am_Amulet_Purple	

10.1.4 Thick and dashed line styles

The following styles are black.

Am_Thin_Line	Am_Line_1	Am_Line_4	Am_Dashed_Line
Am_Line_0	Am_Line_2	Am_Line_8	Am_Dotted_Line

10.1.5 Stippled styles

The following styles are all black transparent or black and white opaque fills

Am_Gray_Stipple	Am_Opaque_Gray_Stipple
Am_Light_Gray_Stipple	Am_Diamond_Stipple
Am_Dark_Gray_Stipple	Am_Opaque_Diamond_Stipple

10.1.6 Am_No_Style

The special Am_Style **Am_No_Style** is used when you do not want an object to have a line or fill style (transparent fill or line style). It can be used interchangeably with NULL.

10.2 Am_Font

10.2.1 Constructors

```
Am_Font (Am_Font_Family_Flag family = Am_FONT_FIXED,
         bool is_bold = false,
         bool is_italic = false,
         bool is_underline = false,
         Am_Font_Size_Flag size = Am_FONT_MEDIUM)

Am_Font (const char* the_name)
```

10.2.2 Predefined Font

Am_Default_Font - a fixed, medium-sized font

10.3 Predefined formula constraints

`Am_Fill_To_Bottom` - Put in an object's `Am_HEIGHT` slot, causes the object to size itself so it's tall enough to fill to the bottom of its owner. `Am_Fill_To_Bottom` leaves a border below the object, with a size equal to the object's `Am_Y_OFFSET` slot.

`Am_Fill_To_Right` - Analogous to `Am_Fill_To_Bottom`, used in the `Am_WIDTH` slot of an object. The `Am_X_OFFSET` slot of the object is used to measure the border to the right of the object.

`Am_Width_Of_Parts` - Useful for computing the width of a group: returns the distance between the group's left and the right of its rightmost part. You might put this into a group's `Am_WIDTH` slot.

`Am_Height_Of_Parts` - Analogous to `Am_Width_Of_Parts`, but for the `Am_HEIGHT` of a group.

`Am_Right_Is_Right_Of_Owner` - Useful for keeping a part at the right of its owner. Put this formula in the `Am_LEFT` slot of the part.

`Am_Bottom_Is_Bottom_Of_Owner` - Useful for keeping a part at the bottom of its owner. Put this formula in the `Am_TOP` slot of the part.

`Am_Center_X_Is_Center_Of_Owner` - Useful for centering a part horizontally within its owner. Put this formula in the `Am_LEFT` slot of the part.

`Am_Center_Y_Is_Center_Of_Owner` - Useful for centering a part vertically within its owner. Put this formula in the `Am_TOP` slot of the part.

`Am_Center_X_Is_Center_Of` - Useful for horizontally centering `obj1` inside `obj2`. Put this formula in the `Am_LEFT` slot of `obj1`, and put `obj2` in the `Am_CENTER_X_OBJ` slot of `obj1`.

`Am_Center_Y_Is_Center_Of` - Useful for vertically centering `obj1` inside `obj2`. Put this formula in the `Am_TOP` slot of `obj1`, and put `obj2` in the `Am_CENTER_Y_OBJ` slot of `obj1`.

`Am_Horizontal_Layout` - Constraint which lays out the parts of a group horizontally in one or more rows. Put this into the `Am_LAYOUT` slot of a group.

`Am_Vertical_Layout` - Constraint which lays out the parts of a group vertically in one or more columns. Put this into the `Am_LAYOUT` slot of a group.

`Am_Same_As (Am_Slot_Key key)` - This slot gets its value from the specified slot (`key`) in the same object. Equivalent to `{ return self.GV(key); }`

`Am_From_Owner (Am_Slot_Key key)` - This slot gets its value from the specified slot (`key`) in the object's owner. Equivalent to `{ return self.GV_Owner().GV(key); }`

`Am_From_Part (Am_Slot_Key part, Am_Slot_Key key)` - This slot gets its value from the specified slot (`key`) in the specified part (`part`) of this object. Equivalent to `{ return self.GV_Part(part).GV(key); }`

10.4 Opal Graphical Objects

10.4.1 Am_Graphical_Object

Slot	Default Value	Type
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	10	int
Am_HEIGHT	10	int
Am_VISIBLE	true	bool

10.4.2 Am_Line

A single straight line segment from `(x1, y1)` to `(x2, y2)`.

Slot	Default Value	Type
Am_LINE_STYLE	Am_Black	Am_Style
Am_X1	0	int
Am_Y1	0	int
Am_X2	0	int
Am_Y2	0	int
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	1	int
Am_HEIGHT	1	int
Am_VISIBLE	true	bool
Am_HIT_THRESHOLD	0	int

Am_X1, Am_Y1, Am_X2, Am_Y2, Am_LEFT, Am_TOP, Am_WIDTH, and Am_HEIGHT are constrained in such a way that if any one of them changes, the rest will automatically be updated to reflect that change.

10.4.3 Am_Rectangle

Slot	Default Value	Type	
Am_VISIBLE	true	bool	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	10	int	
Am_HEIGHT	10	int	
Am_FILL_STYLE	Am_Black	Am_Style	*Inside of rectangle*
Am_LINE_STYLE	Am_Black	Am_Style	*Edge of rectangle*

10.4.4 Am_Arc

Am_Arc is used for circles, ellipses, and arc or pie shaped segments.

Slot	Default Value	Type	
Am_VISIBLE	true	bool	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	10	int	
Am_HEIGHT	10	int	
Am_ANGLE1	0	0..360	*Origin, degrees from 3 o'clock*
Am_ANGLE2	360	0..360	*Terminus, distance from origin*
Am_FILL_STYLE	Am_Black	Am_Style	*Inside of arc*
Am_LINE_STYLE	Am_Black	Am_Style	*Edge of arc*

10.4.5 Am_Roundtangle

A rectangle with rounded corners.

Slot	Default Value	Type	
Am_VISIBLE	true	bool	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	10	int	
Am_HEIGHT	10	int	
Am_RADIUS	Am_SMALL_RADIUS	Am_Radius_Flag or int	*{ Am_SMALL_RADIUS, Am_MEDIUM_RADIUS, Am_LARGE_RADIUS }*
Am_FILL_STYLE	Am_Black	Am_Style	*Inside of roundtangle*
Am_LINE_STYLE	Am_Black	Am_Style	*Edge of roundtangle*

10.4.6 Am_Polygon

A series of connected line segments, optionally filled.

Slot	Default Value	Type
Am_VISIBLE	true	bool
Am_LEFT	<formula>	int
Am_TOP	<formula>	int
Am_WIDTH	<formula>	int
Am_HEIGHT	<formula>	int
Am_POINT_LIST	empty Am_Point_List	Am_Point_List
Am_LINE_STYLE	Am_Black	Am_Style
Am_FILL_STYLE	Am_Black	Am_Style
Am_HIT_THRESHOLD	0	int
Am_WANT_PENDING_DELETE	false	bool
Am_SELECT_OUTLINE_ONLY	0	bool
Am_SELECT_FULL_INTERIOR	0	bool

10.4.7 Am_Text

A single line, single font text object.

Slot	Default Value	Type	
Am_VISIBLE	true	bool	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	<formula>	int	
Am_HEIGHT	<formula>	int	
Am_TEXT	""	Am_String	*String to display*
Am_FONT	Am_Default_Font	Am_Font	
Am_CURSOR_INDEX	Am_NO_CURSOR	int	*Position of cursor in string*
Am_LINE_STYLE	Am_Line_2	Am_Style	*Color of text, cursor*
Am_FILL_STYLE	Am_No_Style	Am_Style	*Background behind text*
Am_X_OFFSET	<formula>	int	
Am_INVERT	false	bool	*Exchanges line and fill style*

10.4.8 Am_Bitmap

An image, either color or monochrome.

Slot	Default Value	Type	
Am_VISIBLE	true	bool	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	<formula>	int	
Am_HEIGHT	<formula>	int	
Am_LINE_STYLE	Am_Black	Am_Style	*Color of on pixels*
Am_FILL_STYLE	Am_No_Style	Am_Style	*Color of off pixels if opaque stipple*
Am_IMAGE	Am_No_Image	Am_Image_Array	*Stipple pattern*

10.4.9 Am_Group

An invisible container object for grouping other graphical objects together.

Slot	Default Value	Type	
Am_VISIBLE	true	bool	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	10	int	
Am_HEIGHT	10	int	
Am_GRAPHICAL_PARTS	empty Am_Value_List	Am_Value_List	*Read only*
Am_X_OFFSET	0	int	
Am_Y_OFFSET	0	int	
Am_H_SPACING	0	int	
Am_V_SPACING	0	int	
Am_H_ALIGN	Am_CENTER_ALIGN	{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}	
Am_V_ALIGN	Am_CENTER_ALIGN	{Am_TOP_ALIGN, Am_BOTTOM_ALIGN, Am_CENTER_ALIGN}	
Am_FIXED_WIDTH	Am_NOT_FIXED_SIZE	int	
Am_FIXED_HEIGHT	Am_NOT_FIXED_SIZE	int	
Am_INDENT	0	int	
Am_MAX_RANK	false	int, bool	
Am_MAX_SIZE	false	int, bool	

10.4.9.1 Am_Resize_Parts_Group

This group has the same slots as an Am_Group, but when resized, it resizes its parts proportionately, instead of simply clipping to the group's bounding box.

10.4.10 Am_Map

A group that creates its parts as many instances of a prototype object.

Slot	Default Value	Type
Am_VISIBLE	true	bool
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	Am_Width_Of_Parts	int
Am_HEIGHT	Am_Height_Of_Parts	int
Am_GRAPHICAL_PARTS	<formula>	Am_Value_List
Am_ITEMS	0	int, Am_Value_List
Am_ITEM_PROTOTYPE	Am_No_Object	Am_Object
Am_LAYOUT	NULL	<formula>
Am_X_OFFSET	0	int
Am_Y_OFFSET	0	int
Am_H_SPACING	0	int
Am_V_SPACING	0	int
Am_H_ALIGN	Am_CENTER_ALIGN	{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}
Am_V_ALIGN	Am_CENTER_ALIGN	{Am_TOP_ALIGN, Am_BOTTOM_ALIGN, Am_CENTER_ALIGN}
Am_FIXED_WIDTH	Am_NOT_FIXED_SIZE	int
Am_FIXED_HEIGHT	Am_NOT_FIXED_SIZE	int
Am_INDENT	0	int
Am_MAX_RANK	false	int, bool
Am_MAX_SIZE	false	int, bool

10.4.11 Am_Window

Slot	Default Value	Type	
Am_VISIBLE	true	bool	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	100	int	
Am_HEIGHT	100	int	
Am_GRAPHICAL_PARTS	empty Am_Value_List	Am_Value_List	*read only*
Am_FILL_STYLE	Am_White	Am_Style	
Am_MAX_WIDTH	0	int	
Am_MAX_HEIGHT	0	int	
Am_MIN_WIDTH	1	int	
Am_MIN_HEIGHT	1	int	
Am_TITLE	"Amulet"	char*	
Am_ICON_TITLE	"Amulet"	char*	
Am_ICONIFIED	false	bool	
Am_USE_MIN_WIDTH	false	bool	
Am_USE_MIN_HEIGHT	false	bool	
Am_USE_MAX_WIDTH	false	bool	
Am_USE_MIN_HEIGHT	false	bool	
Am_QUERY_POSITION	false	bool	
Am_QUERY_SIZE	false	bool	
Am_IS_COLOR	formula	bool	*read only*
Am_OMIT_TITLE_BAR	false	bool	
Am_CLIP_CHILDREN	false	bool	
Am_DESTROY_WINDOW_METHOD	Am_Default_Window_Destroy_Method	Am_Object_Method	
Am_DOUBLE_BUFFER	true	bool	
Am_SAVE_UNDER	false	false	

10.5 Interactors

10.5.1 Am_Interactor

Do not make instances of Am_Interactor. Instead, use the specific interactors described on the following pages. Am_Interactor is used to create custom interactors.

Slot	Default Value	Type	
Am_START_WHEN	Am_Default_ Start_Char	Am_Input_Char	
Am_START_WHERE_TEST	Am_Inter_In_ Object_Or_Part	Am_Where_ Method	
Am_ABORT_WHEN	Am_Input_Char ("CONTROL_g")	Am_Input_Char	
Am_INTER_BEEP_ON_ABORT	true	bool	
Am_RUNNING_WHERE_OBJECT	true	Am_Object, bool	
Am_RUNNING_WHERE_TEST	Am_Inter_In_ Object_Or_Part	Am_Where_ Method	
Am_STOP_WHEN	Am_Default_ Stop_Char	Am_Input_Char	
Am_ACTIVE	true	bool	*Section 5.3.3.3*
Am_START_OBJECT	0	Am_Object	*Section 5.4.1*
Am_START_CHAR	0	Am_Input_Char	*Section 5.4.1*
Am_CURRENT_OBJECT	0	Am_Object	*Section 5.4.1*
Am_RUN_ALSO	false	bool	*Section 5.4.2*
Am_PRIORITY	1.0	float	*Section 5.4.2*
Am_MULTI_OWNERS	NULL	Am_Value_List or NULL	*Section 5.4.3*
Am_MULTI_FEEDBACK_ OWNERS	NULL	Am_Value_List or NULL	*Section 5.4.3*
Am_WINDOW	NULL	Am_Window	*Set with current window*
Am_COMMAND	Am_Command	Am_Command	*Section 5.6*

10.5.2 Am_Choice_Interactor

Slot	Default Value	Type	
All of ths slots of Am_Interactor, with the following changes:			
Am_RUNNING_WHERE_OBJECT	<formula>	Am_Object, bool	*computes owner*
Am_RUNNING_WHERE_TEST	<formula>	Am_Where_Method	*same as start*
Am_HOW_SET	Am_CHOICE_TOGGLE	Am_Choice_How_Set	
Am_FIRST_ONE_ONLY	false	bool	*whether menu- or button-like*
Am_VALUE	NULL	Am_Object or Am_Value_List of objects	

10.5.3 Am_One_Shot_Interactor

Slot	Default Value	Type	
All of ths slots of Am_Interactor, with the following changes:			
Am_START_WHERE_TEST	Am_Inter_In	Am_Where_Method	
Am_RUNNING_WHERE_OBJECT	<formula>	Am_Object, bool	*computes owner*
Am_RUNNING_WHERE_TEST	<formula>	Am_Where_Method	*same as start*
Am_HOW_SET	Am_CHOICE_TOGGLE	Am_Choice_How_Set	
Am_FIRST_ONE_ONLY	false	bool	*whether menu- or button-like*
Am_VALUE	NULL	Am_Object or Am_Value_List of objects	
Am_CONTINUOUS	false	bool	

10.5.4 Am_Text_Edit_Interactor

Slot	Default Value	Type
All of ths slots of Am_Interactor, with the following changes:		
Am_START_WHERE_TEST	Am_Inter_In_Text_ Object_Or_Part	Am_Where_Method
Am_STOP_WHEN	Am_Input_Char("RETURN")	Am_Input_Char
Am_VALUE	""	Am_String
Am_WANT_PENDING_ DELETE	false	bool
Am_TEXT_EDIT_METHOD	Am_Default_Text_ Edit_Method	Am_Text_Edit_ Method
Am_EDIT_TRANSLATION_ TABLE	Am_Edit_Translation_ Table::Default_Table()	Am_Edit_ Translation_Table

10.5.5 Am_Move_Grow_Interactor

Slot	Default Value	Type	
All of ths slots of Am_Interactor, with the following changes:			
Am_GROWING	false	bool	
Am_AS_LINE	<formula>	bool	
Am_FEEDBACK_OBJECT	NULL	Am_Object	*interim feedback*
Am_GRID_X	0	int	
Am_GRID_Y	0	int	
Am_GRID_ORIGIN_X	0	int	
Am_GRID_ORIGIN_Y	0	int	
Am_GRID_METHOD	NULL	Am_Custom_ Gridding_Method	
Am_WHERE_ATTACH	Am_ATTACH_ WHERE_HIT	Am_Move_Grow_ Where_Attach	*Am_ATTACH_.. {WHERE_HIT, NW, N, NE, E, SE, S, SW, W, END_1, END_2, CENTER}*
Am_MINIMUM_WIDTH	0	int	
Am_MINIMUM_HEIGHT	0	int	
Am_MINIMUM_LENGTH	0	int	
Am_VALUE	NULL	Am_Inter_Location	

10.5.6 Am_New_Points_Interactor

Slot	Default Value	Type	
All of ths slots of Am_Interactor, with the following changes:			
Am_AS_LINE	0	bool	
Am_FEEDBACK_OBJECT	NULL	Am_Object	
Am_HOW_MANY_POINTS	2	int	
Am_FLIP_IF_CHANGE_SIDES	true	bool	
Am_ABORT_IF_TOO_SMALL	false	bool	
Am_GRID_X	0	int	
Am_GRID_Y	0	int	
Am_GRID_ORIGIN_X	0	int	
Am_GRID_ORIGIN_Y	0	int	
Am_GRID_METHOD	0	Am_Custom_ Gridding_Method	
Am_MINIMUM_WIDTH	0	int	
Am_MINIMUM_HEIGHT	0	int	
Am_MINIMUM_LENGTH	0	int	
Am_CREATE_NEW_OBJECT_ METHOD	NULL	Am_Create_New_ Object_Method	
Am_START_WHERE_TEST	Am_Inter_In	Am_Where_Method	
Am_VALUE	NULL	Am_Object	*newly created object*

10.5.7 Am_Gesture_Interactor

Slot	Default Value	Type	
All of ths slots of Am_Interactor, with the following changes:			
Am_FEEDBACK_OBJECT	NULL	Am_Object	
Am_CLASSIFIER	NULL	Am_Gesture_ Classifier	
Am_MIN_NONAMBIGUITY_ PROB	0	float	
Am_MAX_DIST_TO_MEAN	0	float	
Am_ITEMS	0	Am_Value_List	
Am_POINT_LIST	Am_Point_List()	Am_Point_List	
Am_START_WHERE_TEST	Am_Inter_In	Am_Where_Method	
Am_VALUE	""	Am_String	*name of gesture*

10.6 Widget objects

10.6.1 Am_Border_Rectangle

A Motif-like rectangle with border.

Slot	Default Value	Type	
Am_SELECTED	false	bool	
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	*{Am_MOTIF_LOOK, Am_MACINTOSH_LOOK, Am_WINDOWS_LOOK }*
Am_WIDTH	50	int	
Am_HEIGHT	50	int	
Am_TOP	0	int	
Am_LEFT	0	int	
Am_VISIBLE	true	bool	
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style	

10.6.2 Am_Button

A single button

Slot	Default Value	Type	
Am_VALUE	NULL	Am_Value	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	<formula>	int	
Am_HEIGHT	<formula>	int	
Am_H_ALIGN	Am_CENTER_ALIGN	{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}	
Am_FIXED_WIDTH	Am_NOT_FIXED_SIZE	int	
Am_FIXED_HEIGHT	Am_NOT_FIXED_SIZE	int	
Am_INDENT	0	int	
Am_MAX_RANK	false	bool	
Am_MAX_SIZE	false	bool	
Am_ITEM_OFFSET	5	int	
Am_ACTIVE	<formula>	bool	
Am_ACTIVE_2	true	bool	
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	*{Am_MOTIF_LOOK, Am_MACINTOSH_LOOK, Am_WINDOWS_LOOK}*
Am_KEY_SELECTED	false	bool	
Am_FONT	Am_Default_Font	Am_Font	
Am_FINAL_FEEDBACK_WANTED	false	bool	
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style	
Am_COMMAND	Am_Command	Am_Command	

10.6.3 Am_Button_Panel

Many related buttons, automatically laid out in a group, with one interactor running all of them.

Slot	Default Value	Type	
Am_VALUE	NULL	Am_Value	
Am_WIDTH	Am_Width_Of_Parts	int	
Am_HEIGHT	Am_Width_Of_Parts	int	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	<formula>	int	*Read-only*
Am_HEIGHT	<formula>	int	*Read-only*
Am_HOW_SET	Am_CHOICE_SET	Am_How_Set	
Am_ITEM_OFFSET	3	int	
Am_ACTIVE	<formula>	bool	
Am_ACTIVE_2	true	bool	
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	*{Am_MOTIF_LOOK, Am_MACINTOSH_LOOK , Am_WINDOWS_LOOK}*
Am_KEY_SELECTED	false	bool	
Am_FONT	Am_Default_Font	Am_Font	
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style	
Am_FINAL_ FEEDBACK_WANTED	false	bool	
Am_LAYOUT	Am_Vertical_ Layout	{Am_Vertical_Layout, Am_Horizontal_Layout, NULL, etc.}	
Am_H_ALIGN	Am_LEFT_ALIGN	{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}	
Am_ITEMS	0	int, Am_Value_List of commands or strings, etc.	
Am_COMMAND	Am_Command	Am_Command	

10.6.4 Am_Radio_Button_Panel: Am_Button_Panel

A group of mutually exclusive radio-button style selection buttons.

Slot	Default Value	Type
all the slots of the button panel, with the following changes		
Am_BOX_WIDTH	15	int
Am_BOX_HEIGHT	15	int
Am_BOX_ON_LEFT	true	bool
Am_FIXED_WIDTH	false	int, bool
Am_FINAL_FEEDBACK_WANTED	true	bool
Am_H_ALIGN	<formula>	{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}

10.6.5 Am_Checkbox_Panel: Am_Button_Panel

A group of checkbox-style selection buttons.

Slot	Default Value	Type
all the slots of the button panel, with the following changes		
Am_HOW_SET	Am_CHOICE_LIST_TOGGLE	Am_How_Set
Am_BOX_WIDTH	15	int
Am_BOX_HEIGHT	15	int
Am_BOX_ON_LEFT	true	bool
Am_FIXED_WIDTH	false	int, bool
Am_FINAL_FEEDBACK_WANTED	true	bool
Am_H_ALIGN	<formula>	{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}

10.6.6 Am_Menu: Am_Button_Panel

A single menu panel

Slot	Default Value	Type
all the slots of the button panel		
Am_FINAL_FEEDBACK_WANTED	false	bool
Am_WIDTH	<formula>	int
Am_HEIGHT	<formula>	int
Am_X_OFFSET	2	int
Am_Y_OFFSET	2	int
Am_V_SPACING	-2	int

10.6.7 Am_Menu_Bar: Am_Menu

A menubar and its submenus.

Slot	Default Value	Type	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	<formula>	int	*width of owner*
Am_HEIGHT	<formula>	int	*height of text in menubar*
Am_ACTIVE	<formula>	bool	
Am_ACTIVE_2	true	bool	
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	
Am_FONT	Am_Default_Font	Am_Font	
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style	
Am_ITEMS	NULL	Am_Value_List	
Am_COMMAND	Am_Command	Am_Command	

10.6.8 Am_Vertical_Scroll_Bar

Slot	Default Value	Type	
Am_VALUE	50	Am_Value	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	20	int	
Am_HEIGHT	200	int	
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style	
Am_VALUE_1	0	int or float	*Value at top*
Am_VALUE_2	100	int or float	*Value at bottom*
Am_SMALL_INCREMENT	1	int or float	*When click arrow*
Am_LARGE_INCREMENT	10	int or float	*When click "page"*
Am_PERCENT_VISIBLE	0.2	float	*Size of indicator*
Am_COMMAND	Am_Command	Am_Command	

10.6.9 Am_Horizontal_Scroll_Bar

Slot	Default Value	Type	
Am_VALUE	50	Am_Value	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	200	int	
Am_HEIGHT	20	int	
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style	
Am_VALUE_1	0	int or float	*Value at left*
Am_VALUE_2	100	int or float	*Value at right*
Am_SMALL_INCREMENT	1	int or float	*When click arrow*
Am_LARGE_INCREMENT	10	int or float	*When click "page"*
Am_PERCENT_VISIBLE	0.2	float	*Size of indicator*
Am_COMMAND	Am_Command	Am_Command	

10.6.10 Am_Scrolling_Group

A group with scroll bars.

Slot	Default Value	Type	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	150	int	
Am_HEIGHT	150	int	
Am_X_OFFSET	0	int	*Where scrolled to*
Am_Y_OFFSET	0	int	*Where scrolled to*
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style	
Am_INNER_FILL_STYLE	0	Am_Style or 0	*If 0 uses FILL_STYLE*
Am_LINE_STYLE	Am_Thin_Line	Am_Style	*Border of scrolling area*
Am_H_SCROLL_BAR	true	bool	*Whether show horiz bar*
Am_V_SCROLL_BAR	true	bool	*Whether show vertical bar*
Am_H_SCROLL_BAR_ON_TOP	false	bool	
Am_V_SCROLL_BAR_ON_LEFT	false	bool	
Am_H_SMALL_INCREMENT	10	int	
Am_H_LARGE_INCREMENT	<formula>	int	*Computed from page size*
Am_V_SMALL_INCREMENT	10	int	
Am_V_LARGE_INCREMENT	<formula>	int	*Computed from page size*
Am_INNER_WIDTH	400	int	*Size of scrollable area*
Am_INNER_HEIGHT	400	int	*Size of scrollable area*

10.6.11 Am_Text_Input_Widget

A labeled text input field, for getting a single line of text from a user.

Slot	Default Value	Type
Am_VALUE	""	Am_String
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	150	int
Am_HEIGHT	<formula>	int
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look
Am_FONT	Am_Default_Font	Am_Font
Am_LABEL_FONT	bold_font	Am_Font
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style
Am_ACTIVE_2	true	bool
Am_COMMAND	Am_Command	Am_Command

10.6.11.1 Am_Tab_To_Next_Widget_Interactor

Add this interactor to a window to allow the user to go from one Am_Text_Input_Widget to the next using the TAB key..

Slot	Default Value	Type	
Am_START_WHEN	Am_Input_Char ("ANY_TAB")	Am_Input_Char	*how starts*
Am_LIST_OF_TEXT_ WIDGETS	<formula>	Am_Value_List of Am_Text_Input_ Widgets	*default is all parts of owner of Interactor that are text widgets*
Am_PRIORITY	105	float or int	*should be larger than running priority (100)*

10.6.12 Am_Selection_Widget

Selection handles used to move and grow standard graphical objects.

Slot	Default Value	Type	
Am_VALUE	NULL	Am_Value_List	*list of selected objects*
Am_START_WHEN	Am_Input_Char ("ANY_LEFT_DOWN")	Am_Input_Char	
Am_FILL_STYLE	Am_Black	Am_Style	*color of handles*
Am_VALUE	Am_Value_List()	Am_Value_List	
Am_ACTIVE	true	bool	
Am_OPERATES_ON	NULL	Am_Object	*group to select from*
Am_START_WHERE_TEST	Am_Inter_In_Part	Am_Where_Method	
Am_COMMAND	Am_Command	Am_Command	*object selected*
Am_MOVE_GROW_COMMAND	Am_Command	Am_Command	*object moved/ grown*

10.6.13 Am_Option_Button

A button that brings up a menu of options to choose from.

Slot	Default Value	Type
Am_VALUE	<formula>	Am_Value
Am_ITEMS	NULL	Am_Value_List
Am_WIDTH	<formula>	int
Am_COMMAND	Am_Command	Am_Command

10.6.14 Am_Alert_Dialog

A dialog box that displays informational text and an "Okay" button.

Slot	Default Value	Type	
Am_TITLE	"Alert"	Am_String	
Am_ICON_TITLE	"Alert"	Am_String	
Am_ITEMS	NULL	Am_Value_List	*Am_Strings to display*
Am_FILL_STYLE	Am_Motif_Gray	Am_Style	
Am_V_SPACING	5	int	
Am_H_SPACING	10	int	
Am_H_ALIGN	Am_CENTER_ALIGN	Am_LEFT_ALIGN, Am_CENTER_ALIGN, Am_RIGHT_ALIGN	
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	

10.6.15 Am_Text_Input_Dialog

A dialog box that displays informational text,a text input widget, and "Okay" and "Cancel" buttons.

Slot	Default Value	Type	
Am_TITLE	"Text Input Dialog"	Am_String	
Am_ICON_TITLE	"Text Input Dialog"	Am_String	
Am_ITEMS	NULL	Am_Value_List	*Am_Strings to display*
Am_VALID_INPUT	true	bool	*Is text input valid?*
Am_VALUE	""	Am_String	*String in text field*
Am_FILL_STYLE	Am_Motif_Gray	Am_Style	
Am_V_SPACING	5	int	
Am_H_SPACING	10	int	
Am_H_ALIGN	Am_CENTER_ALIGN	Am_LEFT_ALIGN, Am_CENTER_ALIGN, Am_RIGHT_ALIGN	
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	

10.6.16 Am_Choice_Dialog

A dialog box that displays informational text and "Okay" and "Cancel" buttons.

Slot	Default Value	Type	
Am_TITLE	"Choice Dialog"	Am_String	
Am_ICON_TITLE	"Choice Dialog"	Am_String	
Am_ITEMS	NULL	Am_Value_List	*Am_Strings to display*
Am_FILL_STYLE	Am_Motif_Gray	Am_Style	
Am_V_SPACING	5	int	
Am_VALUE	""	Am_String	*"Okay" or "Cancel"*
Am_H_SPACING	10	int	
Am_H_ALIGN	Am_CENTER_ALIGN	Am_LEFT_ALIGN, Am_CENTER_ALIGN, Am_RIGHT_ALIGN	
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	

10.7 Command Objects

10.7.1 Am_Command

Use this command object as a prototype for your custom commands.

Slot	Default Value	Type
Am_DO_METHOD	NULL	Am_Object_Method
Am_UNDO_METHOD	NULL	Am_Object_Method
Am_REDO_METHOD	NULL	Am_Object_Method
Am_SELECTIVE_UNDO_ ALLOWED	<formula>	Am_Selective_ Allowed_Method
Am_SELECTIVE_UNDO_ METHOD	NULL	Am_Object_Method
Am_SELECTIVE_REPEAT_ SAME_ALLOWED	<formula>	Am_Selective_ Allowed_Method
Am_SELECTIVE_REPEAT_ SAME_METHOD	NULL	Am_Object_Method
Am_SELECTIVE_REPEAT_ NEW_ALLOWED	<formula>	Am_Selective_New_ Allowed_Method
Am_SELECTIVE_REPEAT_ ON_NEW_METHOD	NULL	Am_Selected_Repeat_ New_Method
Am_LABEL	"A command"	Am_String or Am_Object
Am_SHORT_LABEL	0	Am_String or 0
Am_ACTIVE	true	bool
Am_VALUE	0	any
Am_IMPLEMENTATION_ PARENT	0	Am_Command object

10.7.2 Am_Menu_Line_Command

A menu line command is used to put a horizontal line in an Am_Menu or submenu widget.

Slot	Default Value	Type
Am_LABEL	"Menu_Line_Command"	Am_String
Am_ACTIVE	false	bool
Am_VALUE	NULL	Am_Value

10.7.3 Am_Selection_Widget_Select_All_Command

Select all of the widgets Am_SELECTION_WIDGET is allowed to select.

Slot	Default Value	Type	
Am_SELECTION_WIDGET	NULL	Am_Object	*Must be set!*
Am_ACCELERATOR	Am_Input_Char ("CONTROL_a")	Am_Input_Char	
Am_LABEL	"Select All"	Am_String or Am_Object	
Am_IMPLEMENTATION_ PARENT	<formula>	Am_Command object	*Parent of selection widget's command*
Am_ACTIVE	true	bool	

10.7.4 Am_Graphics_Set_Property_Command

Set the property of object(s) currently selected by Am_SELECTION_WIDGET.

Slot	Default Value	Type	
Am_GET_WIDGET_VALUE_ METHOD	<method>	Am_Get_Widget_Property_ Method	
Am_GET_OBJECT_VALUE_ METHOD	<method>	Am_Get_Object_Property_ Value_Method	
Am_SET_OBJECT_VALUE_ METHOD	<method>	Am_Set_Object_Property_ Value_Method	
Am_SLOT_FOR_VALUE	Am_FILL_STYLE	Am_Slot_Key	
Am_SELECTION_WIDGET	NULL	Am_Selection_Widget	*Must be set!*
Am_LABEL	"Change Property"	Am_String or Am_Object	
Am_IMPLEMENTATION_ PARENT	0	Am_Command object	
Am_ACTIVE	true	bool	

10.7.5 Am_Graphics_Clear_Command

Delete the objects currently selected by Am_SELECTION_WIDGET.

Slot	Default Value	Type	
Am_SELECTION_WIDGET	NULL	Am_Selection_Widget	*Must be set!*
Am_ACCELERATOR	Am_Input_Char ("DELETE")	Am_Input_Char	
Am_LABEL	"Clear"	Am_String or Am_Object	
Am_IMPLEMENTATION_ PARENT	0	Am_Command object	
Am_ACTIVE	<formula>	bool	*active if selection*

10.7.6 Am_Graphics_Clear_All_Command

Delete all of the objects Am_SELECTION_WIDGET is allowed to select.

Slot	Default Value	Type	
Am_SELECTION_WIDGET	NULL	Am_Selection_Widget	*Must be set!*
Am_ACCELERATOR	Am_Input_Char ("CONTROL_DELETE")	Am_Input_Char	
Am_LABEL	"Clear All"	Am_String or Am_Object	
Am_IMPLEMENTATION_ PARENT	0	Am_Command object	
Am_ACTIVE	<formula>	bool	

10.7.7 Am_Graphics_Cut_Command

Cut object(s) currently selected by Am_SELECTION_WIDGET to Am_CLIPBOARD.

Slot	Default Value	Type	
Am_SELECTION_WIDGET	NULL	Am_Selection_Widget	*Must be set!*
Am_ACCELERATOR	Am_Input_Char ("CONTROL_x")	Am_Input_Char	
Am_CLIPBOARD	NULL	Am_Clipboard	*NULL means Am_Global_Clipboard*
Am_LABEL	"Cut"	Am_String or Am_Object	
Am_IMPLEMENTATION_ PARENT	0	Am_Command object	
Am_ACTIVE	<formula>	bool	

10.7.8 Am_Graphics_Copy_Command

Copy object(s) currently selected by Am_SELECTION_WIDGET to Am_CLIPBOARD

Slot	Default Value	Type	
Am_SELECTION_WIDGET	NULL	Am_Selection_Widget	*Must be set!*
Am_ACCELERATOR	Am_Input_Char ("CONTROL_c")	Am_Input_Char	
Am_CLIPBOARD	NULL	Am_Clipboard	*NULL means Am_Global_Clipboard*
Am_ACTIVE	<formula>	bool	
Am_LABEL	"Copy"	Am_String or Am_Object	
Am_IMPLEMENTATION_ PARENT	0	Am_Command object	

10.7.9 Am_Graphics_Paste_Command

Paste object(s) from `Am_CLIPBOARD`.

Slot	Default Value	Type	
`Am_SELECTION_WIDGET`	`NULL`	`Am_Selection_Widget`	*Must be set!*
`Am_ACCELERATOR`	`Am_Input_Char ("CONTROL_y")`	`Am_Input_Char`	
`Am_CLIPBOARD`	`NULL`	`Am_Clipboard`	*NULL meand Am_Global_Clipboard*
`Am_LABEL`	`"Paste"`	`Am_String or Am_Object`	
`Am_VALUE`	`0`	`any`	
`Am_IMPLEMENTATION_PARENT`	`0`	`Am_Command object`	
`Am_ACTIVE`	`<formula>`	`bool`	

10.7.10 Am_Graphics_Duplicate_Command

Duplicate objects selected by `Am_SELECTION_WIDGET`.

Slot	Default Value	Type	
`Am_SELECTION_WIDGET`	`NULL`	`Am_Selection_Widget`	*Must be set!*
`Am_ACCELERATOR`	`Am_Input_Char ("CONTROL_d")`	`Am_Input_Char`	
`Am_LABEL`	`"Duplicate"`	`Am_String or Am_Object`	
`Am_IMPLEMENTATION_PARENT`	`0`	`Am_Command object`	
`Am_ACTIVE`	`<formula>`	`bool`	

10.7.11 Am_Graphics_Group_Command

Create an `Am_Resize_Group` containing the currently selected objects.

Slot	Default Value	Type	
`Am_SELECTION_WIDGET`	`NULL`	`Am_Selection_Widget`	*Must be set!*
`Am_ACCELERATOR`	`Am_Input_Char ("CONTROL_p")`	`Am_Input_Char`	
`Am_LABEL`	`"Group"`	`Am_String or Am_Object`	
`Am_IMPLEMENTATION_PARENT`	`0`	`Am_Command object`	
`Am_ACTIVE`	`<formula>`	`bool`	

10.7.12 Am_Graphics_Ungroup_Command

Ungroup a group created with the `Group_Command`.

Slot	Default Value	Type	
Am_SELECTION_WIDGET	NULL	Am_Selection_Widget	*Must be set!*
Am_ACCELERATOR	Am_Input_Char ("CONTROL_h")	Am_Input_Char	
Am_LABEL	"Ungroup"	Am_String or Am_Object	
Am_IMPLEMENTATION_ PARENT	0	Am_Command object	
Am_ACTIVE	<formula>	bool	

10.7.13 Am_Undo_Command

Undo the last command on the undo stack.

Slot	Default Value	Type	
Am_LABEL	"Undo"	Am_String or Am_Object	
Am_IMPLEMENTATION_ PARENT	0	Am_Command object	
Am_SELECTION_WIDGET	NULL	Am_Selection_Widget	*Optional*
Am_ACCELERATOR	Am_Input_Char ("CONTROL_z")	Am_Input_Char	
Am_ACTIVE	<formula>	bool	

10.7.14 Am_Redo_Command

Redo the last undone command on the undo stack.

Slot	Default Value	Type	
Am_LABEL	"Redo"	Am_String or Am_Object	
Am_IMPLEMENTATION_PARENT	0	Am_Command object	
Am_SELECTION_WIDGET	NULL	Am_Selection_Widget	*Optional*
Am_ACCELERATOR	Am_Input_Char ("CONTROL_Z")	Am_Input_Char	
Am_ACTIVE	<formula>	bool	

10.7.15 Am_Graphics_To_Top_Command

Move the currently selected object(s) to the top of the Z-ordering.

Slot	Default Value	Type	
Am_LABEL	"To_Top"	Am_String or Am_Object	
Am_IMPLEMENTATION_PARENT	0	Am_Command object	
Am_SELECTION_WIDGET	NULL	Am_Selection_Widget	*Must be set!*
Am_ACCELERATOR	Am_Input_Char ("CONTROL_>")	Am_Input_Char	
Am_ACTIVE	<formula>	bool	

10.7.16 Am_Graphics_To_Bottom_Command

Move the currently selected object(s) to the bottom of the Z-ordering.

Slot	Default Value	Type	
Am_LABEL	"To_Bottom"	Am_String or Am_Object	
Am_IMPLEMENTATION_PARENT	0	Am_Command object	
Am_SELECTION_WIDGET	NULL	Am_Selection_Widget	*Must be set!*
Am_ACCELERATOR	Am_Input_Char ("CONTROL_<")	Am_Input_Char	
Am_ACTIVE	<formula>	bool	

10.7.17 Am_Show_Undo_Dialog_Box_Command

Show the selective undo dialog box. You need to `#include UNDO_DIALOG__H` to use this (see Section 1.6).

Slot	Default Value	Type	
Am_LABEL	"Undo/Redo/ Repeat"	Am_String or Am_Object	
Am_IMPLEMENTATION_ PARENT	Am_NOT_USUALLY_ UNDONE	Am_Command object	
Am_UNDO_DIALOG_BOX	NULL	Am_Undo_Dialog_Box	*Must be set!*
Am_ACTIVE	NULL	bool	

10.7.18 Am_Quit_No_Ask_Command

Quit the current application right now by calling `Am_Exit_Main_Event_Loop()`.

Slot	Default Value	Type
Am_LABEL	"Quit"	Am_String or Am_Object
Am_ACCELERATOR	Am_Input_Char ("CONTROL_q")	Am_Input_Char
Am_IMPLEMENTATION_PARENT	0	Am_Command object
Am_ACTIVE	true	bool

10.8 Undo objects

10.8.1 Am_Undo_Handler

Instantiate this object to create custom undo handlers. Normally you should use an Am_Single_Undo_Object

Slot	Default Value	Type
Am_REGISTER_COMMAND	NULL	Am_Register_Command_ Method
Am_UNDO_ALLOWED	NULL	Am_Object or NULL
Am_PERFORM_UNDO	NULL	Am_Object_Method
Am_REDO_ALLOWED	NULL	Am_Object or NULL
Am_PERFORM_REDO	NULL	Am_Object_Method
AM_SELECTIVE_UNDO_ ALLOWED	NULL	Am_Selective_Allowed_ Method
Am_SELECTIVE_UNDO_METHOD	NULL	Am_Object_Method
AM_SELECTIVE_REPEAT_ SAME_ALLOWED	NULL	Am_Selective_Allowed_ Method
AM_SELECTIVE_REPEAT_ SAME_METHOD	NULL	Am_Object_Method
AM_SELECTIVE_REPEAT_ NEW_ALLOWED	NULL	Am_Selective_New_ Allowed_Method
AM_SELECTIVE_REPEAT_ ON_NEW_METHOD	NULL	Am_Object_Method

10.8.2 Am_Single_Undo_Object

Undo or redo only a single command Add thiis as a part to a window.

Slot	Default Value	Type
Am_REGISTER_COMMAND	<method>	Am_Register_ Command_Method
Am_REDO_ALLOWED	<formula>	Am_Object or 0
Am_UNDO_ALLOWED	<formula>	Am_Object or 0
Am_PERFORM_UNDO	<method>	Am_Object_Method
Am_PERFORM_REDO	<method>	Am_Object_Method

10.8.3 Am_Multiple_Undo_Object

Undo multiple commands, and redo only the last undone command. Add this as a part to a window.:

Slot	Default Value	Type
Am_REGISTER_COMMAND	<method>	Am_Register_ Command_Method
Am_UNDO_ALLOWED	<formula>	Am_Object or 0
Am_REDO_ALLOWED	<formula>	Am_Object or 0
Am_PERFORM_UNDO	<method>	Am_Object_Method
Am_PERFORM_REDO	<method>	Am_Object_Method
Am_LAST_UNDONE_COMMAND	0	Am_Command
Am_SELECTIVE_UNDO_METHOD	<method>	Am_Object_Method
Am_SELECTIVE_REPEAT_ SAME_METHOD	<method>	Am_Object_Method
Am_SELECTIVE_REPEAT_ON_NEW_METHOD	<method>	Am_Object_Method
Am_SELECTIVE_UNDO_ALLOWED	<method>	Am_Selective_Allowed_ Method
Am_SELECTIVE_REPEAT_ SAME_ALLOWED	<method>	Am_Selective_Allowed_ Method
Am_SELECTIVE_REPEAT_ NEW_ALLOWED	<method>	Am_Selective_New_ Allowed_Method

10.8.4 Am_Undo_Dialog_Box

This is a complete dialog box used to handle selective undo of various commands. You need to `#include UNDO_DIALOG__H` to use this (see Section 1.6).

Slot	Default Value	Type	
Am_WIDTH	425	int	
Am_HEIGHT	400	int	
Am_TITLE	"Amulet Selective Undo/Redo/Repeat"	Am_String	
Am_FONT	Am_Default_Font	Am_Font	
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style	
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	
Am_UNDO_HANDLER_TO_DISPLAY	NULL	Am_Undo_Handler	*handler to show values for*
Am_SELECTION_WIDGET	NULL	Am_Selection_Widget	
Am_SCROLLING_GROUP	NULL	Am_Scrolling_Group	

11. Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X