CMU - ITC - 058

An integrated authoring environment

Bruce Arne Sherwood
Center for Design of Educational Computing
and Department of Physics
Carnegie-Mellon University
Pittsburgh PA 15213
(phone 412-578-8530)

1985 June 29

The great potential for educational use of powerful advanced-function workstations is currently limited not only by cost, but also by the lack of a congenial authoring environment for educators. For generality and universality, the Unix operating system has been chosen as the software foundation for such workstations. However, the Unix environment and its computer languages were designed for other purposes and do not give much direct support for the major tasks of educational programming (display and interaction). This makes it difficult for non-expert programmers to create educational materials. The purpose of this working paper is to outline what is needed in such an environment, to describe an initial implementation at Carnegie-Mellon University, and to lay the groundwork for considering what should be done next.

Background

Some universities, notably Brown, Carnegie-Mellon, and MIT, are exploring the educational possibilities of advanced-function workstations. These personal computers have been called "3M machines": a million bytes of memory (with virtual memory support), a million pixels on the screen, and a million machine instructions per second. In order to move toward machine independence of educational applications, the use of these advanced-function workstations has been based on the Unix operating system because it is widely available on many different computers. On university campuses Unix has been the software system preferred by computer scientists, due to the rich set of computer science tools which have been created by and for Unix users. Unix itself is written in the computer language C, so Unix tools tend to be oriented toward support of C programs.

In the hands of skilled system programmers, C is an excellent tool for writing system software. It is natural to attempt to write educational applications for workstations in C. However, C has significant disadvantages for this purpose. Many university faculty members have significant programming skills but are not expert professional programmers. In my own C programming and in that of faculty colleagues I have seen a lot of time lost on picky details. Compiling and linking typical educational programs written in C currently takes several minutes, which seriously impedes rapid development. A large part of educational programming consists of displaying text and graphics in complex ways. Like most computer languages, C gives little or no direct support for such displays. Display-oriented subroutine libraries can go a long way toward filling the gap, but problems remain in native C. For example, C supports only standard ASCII characters, and typically a standard C compiler will not permit italic, bold, or foreign-language characters in a source program. If such text is to be displayed, various inconvenient and indirect subterfuges must be employed.

The Information Technology Center (ITC) at C-MU has built a powerful addition to Unix, called "Andrew". Andrew includes a graphics-oriented window manager, a "base editor" library for sophisticated text manipulations, a "layout manager" for associating different procedures with different parts of a window, and the "Grits" subroutine library for manipulating databases. Andrew is, as I will discuss later, an extremely useful set of tools. However, it is difficult for a non-expert programmer to exploit directly the power Andrew offers.

What do we need?

There is a great need for an integrated authoring environment on advanced-function workstations to make it much easier to exploit the power of these remarkable machines for educational applications. Many of us have dreamed of an environment with at least the following properties: incremental compilation to get speed of revision without paying a significant penalty in speed of execution; a good graphics editor which would be tightly coupled to the program and which would automatically take care of scaling in the modern variable-window environment; error diagnostics that would take the author immediately not just to the line but to the position within the line where the mistake occurred; and extensive on-line documentation linked to the authoring environment. The language itself should avoid obscure syntax and represent as directly as possible what will appear on the screen (e.g., centered or italic text should be represented by centered or italic text, not by program directives). The language should be as device-independent as possible yet easy to extend as new hardware possibilities become apparent.

A major benefit of such an environment is that those few unusual faculty who would like to write their own materials could do so, without having to depend completely on programmers. If such a system could be used unaided by individual professors, it would also make programmers and programming teams more productive.

Undoubtedly there are other needs not included in the minimal list above, and I hope that others will suggest extensions to make the environment as complete as possible. One such extension might be in the area of debugging tools, about which I've done little thinking beyond the obvious desire for some kind of step mode and breakpoint capability.

A trial implementation

We are now able to demonstrate an initial implementation of such an environment! It is not complete, but I have already used it to write some little demonstrations and one nearly real program. It will be rather mature by the end of this summer.

I thought it would be feasible to use the powerful Unix and Andrew tools to implement Microtutor, a machine-independent dialect of Tutor, the language of the PLATO computer-based education system. From my involvement with Microtutor in the PLATO project at the University of

Illinois, I knew that this language incorporates most of the important constructs for interactive educational programming, including easy production of graphics, support for diverse kinds of text, rich sequencing facilities, input analysis routines of various kinds, and good calculational capabilities. As will be discussed later in more detail, Microtutor need not do the whole job, but it could serve at least as a much-needed tool for building user interfaces to programs written in other languages such as C or Lisp. Also, the mechanisms used to implement Microtutor could be used to create integrated programming environments for other languages.

Earlier this spring I was delighted to find that it took only three days to build an expression compiler using the Unix tools Lex and Yacc. These tools made it possible to do such work so much faster than had been possible in my earlier PLATO work, that I rashly predicted that I could have the beginnings of an authoring environment by the end of the summer. My estimate was way off. Starting with only the expression compiler I had built earlier, in just ten days I was able to implement the following:

1) About a third of the Microtutor language, including all the major text and graphics display facilities, floating-point variables, and **while** and **for** loops.

2) A text output command which displays all the special forms (italics, bold, large, small, centering, etc.) that the Andrew editor supports. The execution-time display is the same as the source code (as modified by positioning and by specification of margins). This is a major leap forward, since C and other languages are incapable of this directness (though presumably some kind of preprocessor could in principle be created which would overcome this).

3) Incremental compilation. There is no waiting between making a change and seeing the effect, yet execution speed is very fast.

4) Compilation error reporting to the nearest position within a source line.

5) A graphics editor of a novel kind, so tightly integrated with the source code that there is no separate command language necessary to use it.

6) A command new to Microtutor with four arguments to specify whether you want x scaled to the window size, whether you want y scaled, whether you want text size scaled, and whether you want to constrain the scaling to maintain aspect ratio (so that circles don't become ellipses). With one line of code an absolute-coordinate program turns into one which automatically scales and replots appropriately under window changes. The graphics editor fully supports this automatic scaling.

For floating point expressions I currently produce P-codes which on a Sun workstation execute at 50% the speed of compiled C code. I estimate that it would take a day with the help of someone knowledgeable to generate native machine code, which would execute inside the workstation that one happens to be using. I believe that only source code should be kept in permanent storage, to eliminate the administrative burdens of trying to cope with separate compilations for every brand of workstation.

I possess on-line the text for the book "The μtutor Language" written by my wife and me. I believe it will be relatively easy using the Andrew "Grits" database subroutine library to make powerful connections between programs and this textbook.

There are a few crucial additions to Illinois Microtutor in order to exploit the workstation environment, so we need a distinctive name for the dialect. Andrew doesn't yet support the Greek letter mu which we normally used in the name, and the system is written in C, so we will call it C-MU Tutor.

This has been the most exciting experience of my professional life. I am in a state of euphoric shock. What I was able to do by myself in a week in the Andrew environment would have taken my colleagues and me at Illinois many months to do. Unix and Andrew by themselves are not friendly environments for rapid large-scale production of educational materials, but they provide a fantastic environment for building tools.

Thanks to being able to build on UNIX and Andrew, a few weeks will suffice to build what I believe is a viable educational authors' programming language and environment. The benefits of this initial implementation are in themselves great -- authors will be able to begin to use this language productively by late summer. More generally, this is an example of how authoring facilities may be developed quickly and easily by building on UNIX and on Andrew. I will illustrate the power of the current environment (UNIX and Andrew) by describing how C-MU Tutor was developed. Then I will suggest other educationally useful applications that might be developed in similar ways.

How did it happen?

Much of the credit goes to ITC's base editor subroutine library. This is a set of powerful routines for manipulating documents. In C-MU Tutor source code, Andrew document markers keep track of which sections of the program have changed and therefore need recompilation. Pointers and routines make it trivially easy to take the author to the point within a source line where a lexical or compilation error has been detected. The same document data structure is just right for holding P-codes, because the routines for insertion and deletion automatically do memory management. The graphics editor depends on the ease with which new source can be inserted into the body of the program, and on the ability to identify a mouse-selected region of text. The interrupt-driven interaction loop of the Andrew "layout manager" makes it easy to have compilation take place in the background, concurrent with text editing, even though only one process is involved.

Another major support comes from the advanced-function work station, with all its cycles, bytes, and pixels. In previous work I was often forced to think much more about these resources than about the task. In the

PLATO project we were proud of what we did with few cycles and bytes per user, but we paid a big development price. With a powerful personal computer I could write in C rather than assembler, and compilations and links took only about five minutes. At Illinois almost all the work had to be done in assembler (due to lack of target machine memory and speed), which slowed down the development process a great deal. Also, the shortest turnaround to try a new version of the system was about fifteen minutes and often was much longer. Much of the work had to be done at night in order not to interrupt users.

The number of pixels matters because of the special things which become possible when there is ample room on the screen. For the interactive graphics editor it is essential to display both the C-MU Tutor source code and its execution. While building the C-MU Tutor system, it was useful to display several different program files simultaneously.

Here is a particularly striking example of what happened during those ten days, and how the Andrew workstation environment made it possible. As I woke up one morning, I suddenly realized that I could instantly have an integrated graphics editor. I rushed to campus, and in an hour, with fifteen lines of code, I had it. Here's how it works:

In Microtutor display commands, a basic element is of course an x-y coordinate pair. To add a coordinate pair to a source statement, I use the mouse to position the editing caret at the desired spot in the source code. Then I move the mouse to the execution display and click a position there. I take this mouse x-y, generate alphanumeric source code of the form "120,245", drive the base editor "Insert String" routine to place this text into the source code, and then drive the re-execution (and hence re-compilation) of the modified source code. This re-compilation and re-execution is so fast that I can roam quickly around the screen adding more and more x-y pairs to a growing line drawing.

A related powerful technique is to use the mouse to make a selection of text -- put a box around an x-y pair as though one were going to copy it or cut it out. Then click in the execution area. The new x-y coordinate generated from the mouse position replaces the boxed source code, and the screen replots with the change in effect. The selection box stays in place, so one can quickly move around the screen adjusting this one coordinate point of the drawing, which may be the corner of a box, the center of a circle, or any component of the display. The effect is extraordinary, but thanks to the Andrew environment it took less typing to implement this graphics editor than it does to describe it.

This machinery scales the mouse coordinates inversely to the Microtutor-specified scaling, with the result that automatic screen scaling is handled trivially but powerfully by this editor. Note that I'm simply adding or modifying source in standard Microtutor statements, so there is no separate graphics editor command language. Eventually we may put up a menu so that mouse choices can generate the command names.

C-MU Tutor in a larger context

While Microtutor has many of the modern programming structures, its current definition does not include structured data types, although there is a partial substitute in its rather powerful statement-function capability. I expect that even with a complete implementation, with or without the addition of structured data types, there will be many situations which call for other languages (e.g., Lisp). C-MU Tutor might be the language of choice for much of the user interface aspects of such programs. For example, the C-MU project "Dr. Thevenin", an artificial tutor on circuit theory, uses an eclectic combination of Lisp, Ops5, C, Lex, and Yacc. But none of these offer decent support for putting the circuit diagram on the screen. Since C-MU Tutor exploits essentially all the Andrew display capabilities, but packages them in an extremely easy-to-use form, it has much to offer such projects. For these reasons I intend to make sure that C-MU Tutor is callable and can call other languages.

Another use is in the context of C-MU's Glo (Graphical Layout Organizer). An author uses Glo interactively to specify layouts within a window. These rectangular areas are not mere geometrical shapes, but in the Andrew layout manager regime have specific redraw, update, and mouse hit procedures associated with them. It is possible to specify that one Glo layout be a scrollable base editor document, another be an animation displayer, and another be a Unix shell typescript. A new Glo layout type is C-MU Tutor, and one or more layouts can have C-MU Tutor source files associated with them. Because of this environment, it is not necessary that C-MU Tutor itself do everything. It can coexist and collaborate with other processes in a window.

In addition to these mixed uses, I anticipate that many educational applications will be pure C-MU Tutor programs, because of the speed with which such programs can be written and revised, and because it is possible to provide a highly integrated authoring environment around the language.

What next?

I'm excited about C-MU Tutor because it eliminates compilation delays, lets me write fancy text with italics etc. in the source code itself, and because of its other excellent display-generation properties. But consider this: What I have done so easily for C-MU Tutor might be done by others for Basic and Pascal. With a bit of planning, by the end of this summer we might be able to run Basic or Pascal microcomputer programs in a window, for those educational applications for which we have source code. Many people have been concerned about how we can bring along existing applications into the advanced workstation world. Perhaps this is a way to do it, with the existence proof that it has now been done for Microtutor.

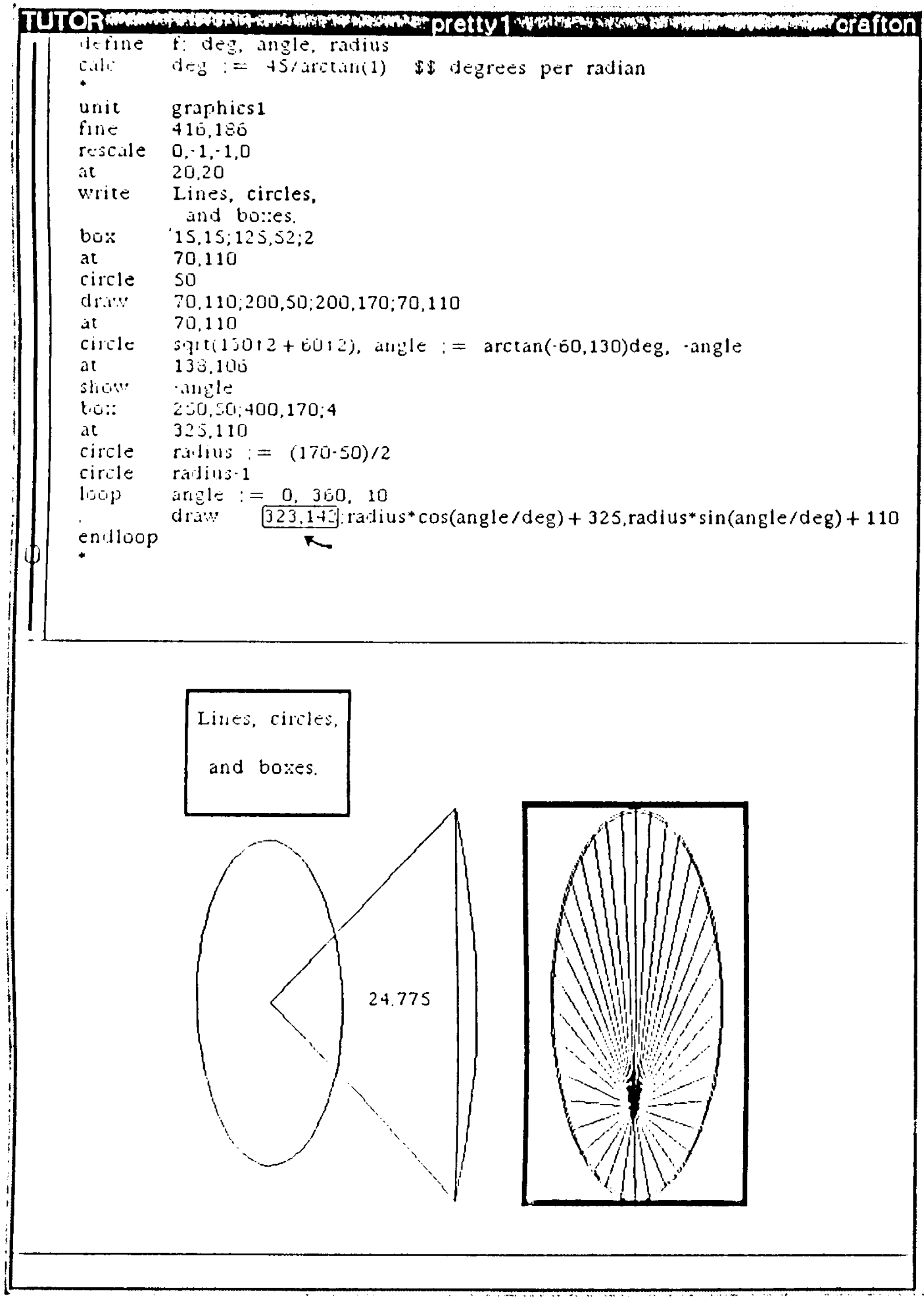
I'm fairly certain about Basic, because it is usually interpreted, and we can interpret faster. I mentioned that at present I semi-compile C-MU Tutor to machine-independent P-codes. The execution of these P-codes is a little slower than the execution of true compiled code, but much faster than interpreting raw source. Given Andrew speed, we can certainly convert Basic source to P-codes on the fly and re-execute these P-codes faster than a microcomputer interprets Basic source. This would give us execution time to handle the highly machine-specific "peeks" and "pokes" people are forced to stick into their Basic programs. We could emulate the effects of these machine-language peeks and pokes by treating them as P-codes on our machine (or by doing the relevant screen manipulations if the peeks and pokes deal with display aspects). I'm less certain about applications originally written in Pascal for the Macintosh, in that a Pascal program compiled on a Lisa and run on a Macintosh executes in native 68000 mode, so we would not have the same speed advantage that exists for interpreted Basic.

A more serious question is whether the Andrew window manager or MIT's X window manager has all the functionality necessary to perform the operations that a Macintosh program does through calling the Macintosh library. My guess is that at the present time we could do most of it if not all. We certainly could build a display inside a window that looks like a Macintosh window. When I get a bit farther along, it might be a useful exercise to try to write a C-MU Tutor program to produce such a display, as a test of the graphics capabilities of C-MU Tutor.

On the other hand, a colleague warns me that compiling the wide range of syntactical structures in Basic, and handling correctly the bizarre peek and poke references to the hardware, may be a much larger task than treating Microtutor. He also suggests that treating Pascal would also be a major task, though I would have thought that Pascal with its regular structures should be straight-forward. However, I don't know these languages nearly as well as I know Microtutor, so he may be right. This is an issue that needs further study. The goal is attractive, however. Educational institutions interested in the Andrew system would be much more interested if they were assured that their own existing programs would run in the new environment.

Steven Lerman, director of Project Athena at MIT, points out that it could be useful to treat C itself in this way. For people who know C and are comfortable with it, this could be a helpful tool. In order to allow fancy Andrew text to appear in the C source code, it would probably not be possible to compile this "almost-C" program with the normal C compiler. But it would let C programmers write in a language very similar to C and get many of the benefits of C-MU Tutor.

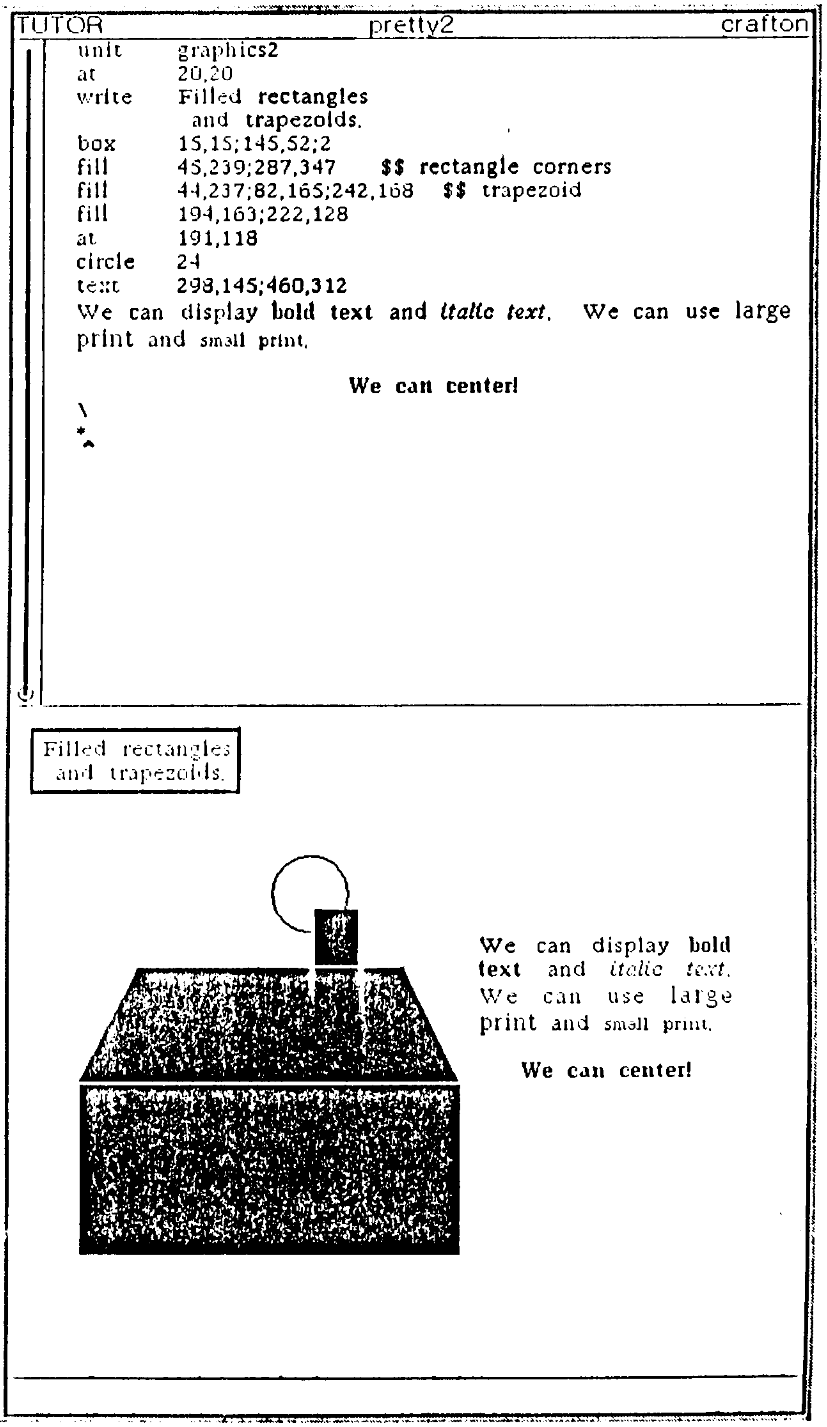
The incremental compiler and graphics editor would be somewhat harder to implement for any of these languages compared with C-MU Tutor, because of the latter's very simple fixed format (including fixed rules for indenting control blocks). Also, there is an important structural difference between C-MU Tutor and these other languages. The Tutor "main-unit" construct provides a natural restart point for re-execution when a window is reshaped, in order to restore the screen display. There is no comparable notion in C or Basic or Pascal. It can however be simulated in C with the "setjmp" routine, which saves the current registers and stack pointer. Even when several levels deep in calls, "longjmp" can be used to restore these registers, effectively restarting the C program at the point where the setjmp routine had been invoked.



TUTOR pretty1 crafton

```
define   f: deg, angle, radius
calc     deg := 45/arctan(1)   $$ degrees per radian
*
unit     graphics1
fine     416,186
rescale  0,-1,-1,0
at       20,20
write    Lines, circles,
          and boxes.
box      15,15;125,52;2
at       70,110
circle   50
draw     70,110;200,50;200,170;70,110
at       70,110
circle   sqrt(150↑2 + 60↑2), angle := arctan(-60,130)deg, -angle
at       138,106
show     -angle
box      250,50;400,170;4
at       325,110
circle   radius := (170-50)/2
circle   radius-1
loop     angle := 0, 360, 10
.        draw     323,143;radius*cos(angle/deg) + 325,radius*sin(angle/deg) + 110
endloop
*
```

Lines, circles,
and boxes.
24.775



TUTOR pretty2 crafton

```
unit     graphics2
at       20,20
write    Filled rectangles
          and trapezoids.
box      15,15;145,52;2
fill     45,239;287,347     $$ rectangle corners
fill     44,237;82,165;242,168   $$ trapezoid
fill     194,163;222,128
at       191,118
circle   24
text     298,145;460,312
We can display bold text and italic text.  We can use large
print and small print.

                    We can center!
\
*
```

Filled rectangles
and trapezoids.
We can display **bold text** and *italic text*. We can use large print and small print.
We can center!