University of Illinois at Urbana-Champaign
Department of Computer Science

MP1: Tutorials and Beginning C++

CS 225 Data Structures
Spring Semester, 1999

Handed out: Friday, January 29, 1999
Due date: Thursday, February 4, 1999 at 11:59 PM

1 Introduction

In this MP, you will accomplish a number of things. First, you will run through quick tutorials of three software tools that you will find useful throughout the semester. Then, for the actual coding part of the MP, you will review some beginning C++ concepts and write some some code yourself that deals with these ideas. You will write a few classes and deal with testing them via a driver file.

Don't be alarmed by the size of this handout – there's a lot of writing, but it's only detailed descriptions of what you have to do. The individual sections shouldn't take you too long. The MP has intentionally been designed to be easy so that you have time to look through the code and slowly review the C++ ideas.

2 Part 1: Programming tool tutorials

The first half of this MP involves familiarizing yourself with certain programming tools that you can make use of on the sparcs. First, you will learn how to use a *Makefile* to manage the compiling of large projects and save yourself compilation time. Second, you will use a program called *purify* to track down memory leaks and other memory errors in your program (you may have used such a tool in CS223 as well). Finally, you will run through a small tutorial on *workshop*, the new Sun IDE that has as one of its tools an extremely useful debugger.

Before you start this tutorial, you want to copy the tutorial code into your own directory. Do this by changing to the directory where you want to deal with your tutorial code and then running the command:

```
cp -R ~cs225/src/tutorial .
```

In case you haven't seen it before, the -R means that the directory "tutorial" and *everything* in it will be copied to a new directory called "tutorial" that is a subdirectory of wherever you are now. Don't forget the period at the end!!!

2.1 `make` and the `Makefile`

NOTE: In this course, we will be using Sun's compiler, `CC`. However, the idea of a `Makefile` works just as well no matter which UNIX compiler you choose to use.

2.1.1 The purpose of the `Makefile`

If you have never used a Makefile before, then your experience compiling from the command line at most consists of repeatedly typing in lines like the following:

```
CC -c file1.C
CC -c file2.C
     .
     .
     .
```

and assorted other command-line manipulations of individual files. Now, certainly, you could write a script to automatically compile all your files and link them, but then everything would be re-compiled each time you ran the script.

The command `make` is designed to avoid that problem. With `make`, you can easily specify a complex compilation, and simply by typing "`make`" at the command line, have that compilation kick into effect for you. All you need to do is specify some of the compilation details in a file known as a `Makefile`.

There is another benefit, too, however: each time you type "`make`", the system will *only* recompile:

1. The files that you have changed
2. The files that use interfaces that have been re-compiled

For example, suppose you coded a `String` class – with files `string.h` and `string.C` – using an `Array` implementation coded in files `array.h` and `array.C`. Certainly, the `String` files would `#include array.h`, because the `String` files are using the `Array` interface functions.

Now, suppose you had a program that used these files, and decided after it was done that you wanted to change some of them. If you made a change to the `Array` implementation, but not the interface, then only `array.C` would be changed. Now, if you were to re-type `make`, only array.C would be recompiled, and then the executable would be linked. Similarly, if all you had changed was `string.C`, then the `make` command would only recompile `string.C` and then re-link. However, if you make a change to `array.h`, then not only would the object file for `Array` have to be recompiled, but also the object file for `String` – since `String` makes use of the `Array` interface, a change in the interface of `Array` potentially affects the compilation of `String` as well, and therefore `String` would need to be recompiled.

However, a file that was unaffected by a change in the `Array` interface would not need to be recompiled, and therefore would not be recompiled (if you set up your `Makefile` correctly).

But, the point is that, once you set up the `Makefile`, all of that is automatically handled whenever you type "`make`".

2.1.2 The `Makefile` explained

Inside the `tutorial` directory you have copied, examine the `Makefile`.. To begin with, you will notice helpful directions throughout, all starting on lines that begin with the pound sign (#). In a `Makefile`, the pound sign is the comment indicator.

Now, look at the first two non-commented lines of the `Makefile`:

```
OBJS = \
        tutfns.o main.o
```

At the start of our `Makefile`, it will be helpful to have a macro that stands for all of the object files needed by the linker. That's what this line is. We have two separate object files, and so we have two separate files listed on the line above. In larger programs, where there are many more files, there will be many more object files listed above.

Two things to take note of: First, the slash after the OBJS simply means, "continue on next line". So, the above is really all one line, and if we had many object files and wanted to write them over three or four lines, we would simply put those same slashes at the end of all but the last line, in order to tie all the object files into the same "one-line" macro. Second, the indentation on the second line is NOT eight spaces; it is a tab. *You have to hit the tab key!* If you use spaces, it will be an error and the `Makefile` will not work.

Next, note that we have a few more macros on the next number of lines. We define a name for the executable file, which you can change if you prefer using `mp0` or `jason` or whatever else instead of `a.out`. We also have macros for the compiler and linker and their options. In particular, note that `CC` is the compiler listed, and if we were using `g++` instead, then `g++` would be listed there instead. Also, note the `-g` on the CCOPTS line. The symbol combination `-g` is a debugging flag, and it will compile the program with debugging information that can be used by a debugger. However, this adds to the size of the final executable, which is why this is an option rather than always automatically done. If you worked for a company, you would probably want the debugging information to be there when you were developing the code, but you would probably want to compile without it before shipping so that you could ship a program that needed less memory.

Finally, we come to the bottom section, where the dependency specification takes place. Ignore the first few groups of statements for the moment, and move down to the lines that read:

```
tutfns.o : tutfns.h tutfns.C
        $(CC) -c $(CCOPTS) tutfns.C
```

The first line says that the object file `tutfns.o` depends on implementation code from `tutfns.C` and interface code from `tutfns.h`. In general, you will have an object file on that line, followed by a colon, followed by all the files that object file depends on for its own correct compilation.

The second line specifies how to build this object file. Here, to build `tutfns.o` we simply compile `tutfns.C`. However, notice that by using `$()` we are invoking one of our previously defined macros. So, the second line is read by the system as:

```
CC -c -g tutfns.C
```

Finally, notice that, again, that second line *must be tabbed*, and that your `Makefile` will not work if you use spaces instead.

Look now at the next two lines, the lines for the `main.o` executable. We see that the dependency line for `main.o` has two files: the `main.C` file and also `tutfns.h`. If you look through the `main.C`, you will notice that `main.C` makes use of functions declared in `tutfns.h` and defined in `tutfns.C`. So, if `tutfns.h` was changed, that would mean that – potentially – the code in `main.C` no longer works (for example, what if we eliminated the functions in the `tutfns.h` file entirely?) So, since `main.C` uses the interface or the set of function headers that `tutfns.h` provides, if the `tutfns.h` file changed, then that means the collection of function headers being offered to `main.C` has changed,

and so `main.o` should be recompiled as well, just so that we can be sure it will still work. Hence, `tutfns.h` must appear on the dependency line for `main.o`. (You *could* leave it out, but you would then end up with compilation and linker errors if you *did* change `tutfns.h`.)

When you first type "`make`", the system will attepmt to create an executable by running the following lines:

```
$(EXENAME):  $(OBJS)
        $(LINK) $(LINKOPTS) $(OBJS)
```

which, of course, expand to:

```
a.out: tutfns.o main.o
        CC -L/opt/SUNWspro/SC4.0/lib tutfns.o main.C
```

In order to do the linking step, it is necessary for the object files to be generated, and thus they are, using the lines we discussed above and leaving alone any currently-existing object file that does not need to be re-compiled. Then, once the object files are generated, the compiler links them all together and creates the executable.

Finally, note the lines:

```
clean:
        -rm *.o $(EXENAME)
        -rm -r Templates.DB
```

Here, we are specifying a particular command for `make`. When we typed `make`, we could just as well have typed `make a.out`. We don't need to, because in the absence of any extra command line name, the system assumes we mean the executable. However, we could also have compiled individual object files by typing lines such as `make asserts.o`. Just like we can do that, we can run the lines listed after "`clean`" above by typing `make clean`.

These lines remove *all* object files, the executable files, and the template database (`CC` creates a template database whenever it compiles templates with your program). What you are left with is just your source code – as if you had never compiled in the first place. This is helpful whenever you want to do a fresh compile, from scratch. Why would you want to do this? Well, there are a few reasons. First, errors in the `Makefile` can result in problems, such as when dependency files are changed but the `Makefile` isn't coded to worry about those changes. In that case, once you fix the `Makefile` you generally want to recompile from scratch, and so you first run `make clean` to clean away the old machine code. Second, sometimes there are problems with the compiler itself – an odd bug that no one has caught, for example. Usually, all the bug consists of is an inability to handle a code revision properly, resulting in bizarre linker errors that are similar to what you would get if you mis-coded your `Makefile`. So, if you are ever getting a *really* bizarre linker error or set of linker errors, and you don't know what is causing them, sometimes what you should try is simply running `make clean` and then recompiling from scratch. By allowing the compiler to re-read the code instead of trying to keep up with dependency issues, you often get around the particular problem things are stuck on.

This is not to say that most of your errors will be like this; quite to the contrary, most of your errors will be of your own doing. But, `make clean` is there for the times when you need it, or at least when you think you *might* need it.

That, then, is an explanation of the `Makefile`. Throughout the semester, you will see different `Makefile` files, but the ideas are always the same as those described above. You can get even *more* complicated, though. If you want to learn about `make` in depth, we recommend you look for a comprehensive reference on the matter (I believe O'Reilly has a small "Nutshell' book on `make`).

2.1.3 Using `make`

Now, while in your `tutorial` directory, simply type "`make`" and hit return. You will see compilation begin, and then stop, once an error is detected. The error is that `main.C` is using a variable called `Debugrray` which is not defined. So, open `main.C` using your choice of editor, and fix line 4 so that instead of reading

```
/* 4  */   AssignMore(Debugrray, 4);
```

it instead reads

```
/* 4  */   AssignMore(DebugArray, 4);
```

That is, add the capital 'A' to correct the spelling mistake. Now, rerun `make`, and this time the program should compile correctly.

That's it! You have now completed the `Makefile` tutorial.

3 Using Sun's IDE, `workshop`

Use the enclosed `Makefile` to compile your code, and then run your compiled code by typing `a.out`. You should get output that looks as follows:

```
We are now demonstrating the debugger.

*** properly allocated array
4
5
6
7
8
9
10

*** improperly allocated array
Segmentation fault
```

and in your tutorial directory there should now be a `core` file.

Segmentation faults happen when you are trying to access the object that a pointer points to, and it turns out that that pointer does not point to what you thought it did. Commonly it instead points to NULL.

You could certainly track this problem down "by eye" – and in fact, this program is simple enough that you may have figured it out already – but we will make use of a debugger to help us find this problem.

While still in the tutorial directory, type the command

```
workshop&
```

You will have to wait a bit, but then a small, long rectangular box will appear, with various symbols on it. In addition, a large box will appear, with options such as "Product Versions" and "Close". For now, click "Close" in that box. We will only be dealing with the smaller box.

In the smaller box, hold the mouse cursor over the icon of the fly in a red circle with a line through it. Notice that on the bottom of the box a description appears that says this is the debug button. Similarly, you can read the descriptions of the other buttons as well. For now, click on the debug icon, since we will learn how to use the debugger.

A window will appear, listing the contents of your current directory. Select `a.out`, so that "a.out" appears in the "Name" window. Then, click on OK to load the debugger with the `a.out` file. Once you do that, there will be some processing and such...just wait while that goes on. Eventually, two windows will appear: the first will have spaces marked such as "Data History" and "Stack", and the second window will be the actual main.C file itself.

The smaller window is where you can control the reading of certain data values, and the larger window is where you can observe the running of the code itself. To begin with, in either window, click on the arrow that points downward and has a thick line on its other end (it will stay "Start" when you hover the mouse over it). A third window (an output window) will pull up, you will see some familar output, and then a small window talking about a "SIGSEGV" will appear. This is just a signal that a segmentation fault has been encountered. This means you will need to step through the program to figure out where the error is. So, first, click on "Dismiss" in that small window. (You might also need to move the windows around a bit to make the output window visible).

Now, move over to the code window. You will notice that the program has stopped on a line in the `AssignMore`. We will see exactly where this is being called from. To do this, we need access to the main() function again, which means we need to move *up* the stack of function calls. Find the icon that shows a colored stack with an arrow pointing up (it is in the code window). Click on this button once, and the debugger will jump *out* of the current function call, and *up* the function stack to the function that called this one. That function is main(), which is where we want to be.

Now, click on the "1" in the comment at the start of line 1. Actually, you can click anywhere at the start of the line; the only intention is to mark that as the spot where you want to add a "stop" or "breakpoint". Now, note that the second icon from the left is a stop sign with an arrow in it (labelled "Stop at" when you hover over it). Click on this, and after a moment, a red stop sign will appear at the start of line 1. Now, whenever we run this program, execution will stop at line 1 and wait for us to signal what to do next.

Now, go back and click the start arrow again. This time, there is a green arrow that pauses on line 1, because that is where we placed our stop point. It is now possible to advance line by line, waiting for a bad effect to occur. To do this, click on the icon with a straight arrow that points to the right. This icon is the fifth from the right, and is labelled "Step Over". Click on this three times, and watch the output window as you click. Note that the green arrow – which marks the next line to be executed – moves down line-by-line, and the output appears line-by-line as well. What you are doing here is running the program line-by-line, by stepping over the current line and executing all of its effects.

What exactly do we mean by "step over"? Well, that phrase is in contrast to "step into". If we are on a line that involves a function call, choosing "step over" will simply run that function and generate the results. Remember, choosing "step over" executes *the current line*, and everything involved with the current line. On the other hand, selecting "Step into" (to the left of "Step over, the green arrow that is bent and points to the right) will *not* step over the line but will instead step into whatever function call is on that line, if any. Since after three clicks of "Step over", you should be on a line that calls `AssignMore`, you can now choose "Step into" to step into that function.

When you do this, `main.C` will disappear from the window, and `tutfns.C` will pull up instead. You will be positioned at the first code line of the `AssignMore` function. This is a two-line function, and you can step through this function just like you stepped through main.C – namely, by using

the "Step over" command to move down line-by-line. Note that the second click of "Step over" will bring you back to the second line again – or rather, *keep* you on the second line, since that's where you were after the first click. This just means that the loop condition was checked, and it was found that you are not supposed to exit the for-loop yet, and so you are placed at the first line of the for-loop's executable code again. Since there is *only* one line of executable code for the for-loop, you are simply kept on the line you were at before. The difference is that now you are about to run this line for the second time, instead of the first.

So, clock on "Step over" six more times, so that you run through the line six more times (remember, you are *about* to run it the second time; you have not done so yet – so you do indeed need to run the line six more times to total seven). At this point, the debugger will exit the for-loop and position itself to run the next line, which in this case is the closing bracket at the end of the function. Choosing "Step over" one more time will bring you back to `main.C`, and the line right after the call to `AssignMore`.

Now, click "Step into" again. You are able to jump into the `Printing` functioon the same way you jumped into the `AssignMore` function. However, now that you are in the function, perhaps you decide that that really isn't what you want to do (or perhaps it could have been a long function, and you only wanted to see the results of the first few lines), and you would prefer to, in one click, run the rest of this function and jump back to the function that called this one. That can be done easily, by using the "Step out" command (the other bent arrow, to the right of "Step over"). Now, you are back in `main.C` and have been through the entire `Printing` call, and are now on line 6 of the the loop a second time, back on the assign line to execute that line of code a third time. (The "Step out" command can prove to be helpful at some times when you have non-reference objects as parameters; since those parameters will be copied using the copy constructor, each function call will call the copy constructors of all our parameters. Since we often don't want to waste time running through the copy constructors (unless we suspect those are what might be causing the problems in our code), we can step out of those copy constructors as soon as we step into them, and once we have done the "step in, step out" sequence for each of our parameters, at that point the next "step in" will actually bring us to the function we are trying to call. This is only a problem, though, when we pass objects of classes we have written; the system is not going to send us into an integer or character copy constructor.)

Now, at this point, you might want a little proof that the `AssignMore` loop worked correctly. To do this, we want to go back into the `AssignMore` function, so click on "Start" again to re-run the program from the beginning. And, once again, click on "Step over" until you reach the `AssignMore` function, and then "Step into" the `AssignMore` function.

Now, move to the "Workshop Debugging" window, the one with the "Data History" and "Stack" windows within it. The "Expression" window allows you to check the values of variables in your program. For example, type the variable name "theArray" into the "Expression" window, and then click on "evaluate". You will see a memory address (a number starting with 0x) appear in the "Data History" window. If you click on "Display" instead of "Evalutate", a new window will appear, with the same memory address but with that address highlighted. When you display memory addresses in these new windows using the "Display" command, the memory addresses will be highlighted and it is possible to click on those addresses to learn what is located there. If you move to the new, small window and click on the memory address that is indicated to be the address stored in `theArray`, a new value will appear in the window. This is likely to be some large (garbage) number. Specifically, you are likely to see something like the following:

```
  *‘a.out‘tutfns.C‘AssignMore‘theArray
 = 142280
```

The variable `theArray` holds a pointer to the first value in the array, and since you have not assigned that value yet, you have garbage data in the memory location. If you return to the "Expression" window and type `theArray[0]`, and then select "Display", you will see displayed the same value that was listed as the value at the location held by `theArray` (in the case of this example, the debugger will tell you that `theArray[0]` holds 1442280).

You can do this for any variable in your program, as long as the variable is in the current scope. For example, even though `main.C` has a variable called `DebugArray`, you cannot display its contents while at this point in the program, because that variable name is not in the current scope (try it and see!). However, if you choose the "Up" button, you can then display the contents of `DebugArray`, since you will then be in the scope where that variable is defined. If you choose to try this, be sure and select the "Down" button (located to the right of the "Up" button) in order to move back into the scope of the `AssignMore` function before moving on with the tutorial.

So, if we want to verify the first few steps of our for-loop, we can do that the same way – by checking the variables in our program. First, choose "Step over" once, to run the for-loop setup line and advance to the array assigning line. Now, if you check the value of `i`, you will see that it is 0. If you now click on "Step over" once more, and then check the value of both `theArray[0]`, and also the value at the address stored in `theArray`, you will see that they both hold the value 4. Another click of "Step over" will change the value of `i` to 1. And so on. That is how you can check variable values inside the debugger.

Now, we want to move on with our check of the error, so click on "Step out" to move out of `AssignMore` and on to line 5 in `main.C`. Now, keep clicking on "Step over" until the segmentation fault is reached again. Notice that it occurs when you try to run line 8, which makes sense, since the `AssignMore` function was where our segmentation fault occured earlier.

So, again click on "Dismiss" in the choice window and then restart the program with "Start". This time, while we are sitting on line 1, click on line 8 and then on "Stop at" again. This will set a second breakpoint, but this one on line 8. Now, it is possible to jump from wherever we are located – line 1 in this case – to the next existing breakpoint that has been set – in this case, line 8 – by using the "Go" command. This command is activated with the button that has a downward pointing arrow (like "Start") but with *no* thick line at the top. It is located to the left of the "Step into" button. Clicking on "Go" will jump us immediately to line 8, and then you can use "Step into" to step into the `AssignMore` function.

Now, before you do anything else, check the value of `theArray`. You will see that it is (nil). This means that our pointer holds the address 0, i.e. NULL. Therefore, any attempt to assign to a value in this array will result in a segmentation fault, because we don't actually have any memory allocated for this array – instead, we just hold the value NULL. You can see this for yourself by clicking on "Step over" twice – the second click, which will run the first assignment, will result in the segmentation fault you have been getting.

Now, click on "Dismiss" in the choice window, and then choose "Up" to view `main.C`. Lo and behold, on line 7 we assigned the pointer `DebugArray2` to NULL instead of to newly-allocated memory. So, that is our error. You can quit the debugger by going back to the tiny window that was the first window that appeared when you started the debugger. Under the "Workshop" menu, choose "Exit workshop".

Finally, to fix the program, open main.C in your editor, and change line 7,

```
/* 7  */   int* DebugArray2 = NULL;
```

to the following

```
/* 7  */   int* DebugArray2 = new int[7];
```

thus allocating the array. Now, if you exit main.C, run "make", and then type "a.out" to run the program, the program should run fine and exit gracefully.

This only scratches the surface of what you can do with the IDE. We encourage you to explore the features of it, including the ability it has to re-compile your code from within the IDE (using "Build"), and the ability to use the debugger code window as an editor. There are help pages built in (under "Help") that can provide you with explanations of `workshop` features, and there is other help available as well. You can also ask us – we haven't used all the features ourselves, either, but we can help you understand a feature you are having trouble with and perhaps – if we haven't used that feature yet – learn a new feature ourselves in the process!

4 `purify` – a memory leak detector

Finally, we are going to take a quick look at a tool called `purify` that will help us detect *memory leaks* – a problem in the code where we have allocated dynamic memory but not deleted it later on. `purify` can help us with other memory problems as well, but memory leaks will be the most common ones you encounter. (You may have used this in CS223 as well.)

To use `purify`, you need to alter the `Makefile`, so open it up in your editor. `purify` is run in the linking phase, not the compiling phase, so we want to add the purify command to the link phase. That is, change this:

```
$(EXENAME):  $(OBJS)
        $(LINK) $(LINKOPTS) $(OBJS)
```

to this:

```
$(EXENAME):  $(OBJS)
        purify $(LINK) $(LINKOPTS) $(OBJS)
```

Again, keep in mind that before the word "purify", you *have a tab, not a bunch of spaces!!!*

Now, just typing `make` will be sufficient to invoke `purify`. Note, though, that since `make` won't do *anything* if the `a.out` file is up to date (no changes to the source files since the last compilation), you will want to `rm a.out` so that `make` will recognize the need for re-linking, and thus invoke `purify`.

*****

NOTE: if you were not using a `Makefile`, you would simply write the word `purify` before your command line compilation, as follows:

```
purify CC main.C tutfns.C
```

or, alternatively,

```
CC -c main.C
CC -c tutfns.C
purify CC main.o tutfns.o
```

*****

When you relink using purify, you wil notice some rather odd, rather long object files being generated. That is okay. Once a new `a.out` has been generated, then you can run the program just as you did before. The difference is, this time a new window will pull up detailing the memory problems with the program.

Take a look at this window. The little arrows are open-close arrows; clicking on them exposes more information (if it swings from pointing across to pointing down), or hids it (if it swings the other way). Click on the "Memory leaked" arrow. This exposes a description of the memory that purify has said we are leaking. The third arrow that appears is labelled "Purify Heap Analysis", and contains some summary information, but the most useful info can be found if you expand the other two arrows, which are both labelled "MLK" (which is, of course, short for "memory leak").

If you first click on the top MLK arrow, and then, second, click on the `main` arrow that appears when you make the first click, you will see a snippet of code with an arrow pointing to line 3. Purify is indicating that this is the leaked memory – that is, the memory allocated on this line is never deleted. If you go back and do the same for the *second* MLK arrow, you will see that that leak occurs on line 7, which was the other allocation of dynamic memory in our short program.

So, now that we know where the memory is leaking, the goal is to figure out what to do about it. And, the solution is generally to add the appropriate delete lines (there might be times, as well, when you are allocating memory you shouldn't have allocated in the first place, and so the correct solution in some of those cases might simply be to remove the lines with "new" rather than to add lines with "delete"). Remember, for each use of `new` to allocate memory, you need to have a corresponding `delete` to get rid of that memory. And, likewise, for each use of `new[]` to allocate an array, you need to use `delete[]` to deallocate that memory.

So, close up the arrows you opened, by clicking on them again. Then, by clicking on the "Finished a.out" arrow at the top of the window, you can close up this entire run of the program. The next run will generate new data (as you will see).

Now, open `main.C` again in your editor, and between lines 9 and 10, add the lines:

```
delete[] DebugArray;
delete[] DebugArray2;
```

.

Then, save, re-make, and re-run. You will notice a second run of data in the purify window, and this time, the "memory leaked" line will indicate that no memory was leaked. If you click on that arrow, the only line there will be the "Purify Heap Analysis" line – there are no errors to report. Thus, you the program is no longer leaking memory! You can close the purify window simply by choosing Exit in the File menu.

You can have a number of memory error messages generated by `purify`, depending on the type of memory errors you have. You can read about them, and read more about `purify` in general, on the man page for `purify`. Just type "man purify". Things like writing beyond the bounds of an array, reading beyond the bounds of an array, reading freed (deleted) memory, and failed memory allocations can all be detected by purify, and the program can make these errors known to you so that you can track their causes down in your program and correct them.

5 The programming assignment

To begin with, you will need to copy the given files into your own directory. The files are located in:

```
~cs225/src/mp1        (MP code)
~cs225/src/mp1/test   (test files)
```

5.1 The given code

The first part of the coding assignment involves reviewing looking over the given code. The topics covered on this MP, either in the given code, or what you have to write, or both, are: the given code are:

```
classes and member functions
easy constructors
arrays that are allocated from the stack
passing arrays to functions
dynamically allocated arrays and objects
if-else and for statements
deleting dynamically allocated arrays and
   objects
getting a program working -- compilation,
   include statements, and so on.
```

Most of it we have covered between lecture and section, but there may be a few details you need to look up in your C++ text, either to understand the code or to complete the assignment. You are welcome to ask on the newsgroup about any syntax concept you don't understand.

The given files are described as follows.

The classes `Fill`, `Cross`, and `Bend` are three of the five "tube shape" classes used by this MP. The other two – `Tee` and `Straight` – will be written by you.

The idea is basically to imagine a grid in which each square contains one of five types of "tube shapes":

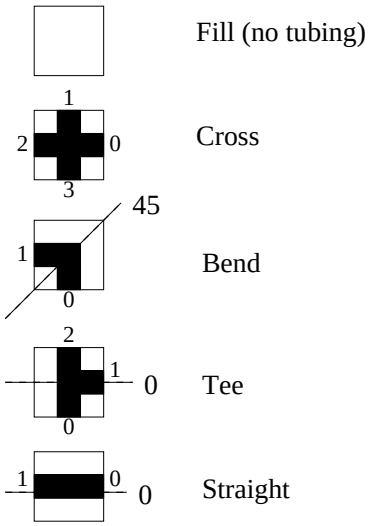

Figure 1: The five tube shapes in this MP

The tube shapes are pre-fabricated tunnel pieces. Some construction companies are in the process of building a city, and the first thing they are doing is setting up tunnels and water and sewer lines underneath the city. They order large cubes of concrete (seen as squares in two dimensions), which have one of the tube patters cut through them. Then, by placing these concrete blocks next to each other, tunnels can be established by lining up tube entrances of two adjacent blocks. (We'll see a diagram later.)

The class `Fill` models a block of concrete with no tubing in it. This would be a space filler block, to be used in a particular spot where there was no need for any tunnel right there. Since there are no tubes, there are no entrances and no orientation. If we want to print out a picture of this block, we can use a 3-by-3 grid of asterisks:

```
***
***
***
```

The class `Cross` models a block of concrete with a cross-shaped tube in it. So, there are four entrances, all of which meet in the middle, allowing people or water to enter from one side of the block and leave through any of the other three sides. No matter how you rotate this shape, as long as the base lies flat the picture looks identical, and so there is no concept of orientation for a `Cross` block. Therefore, we don't allow it to be rotated. If we number the entrances as shown, we will always place a `Cross` block in the ground so that entrance number 1 faces the "top" of our blueprint. This is also how it appears in the above figure. This block can be drawn by using dots to illustrate the open spaces in the block:

```
*.*
...
*.*
```

The other three classes likewise model such blocks, and their tubings are as in the above picture. `Bend` has two entrances, and has a 90 degree turn in the middle. Because of this, it is possible to turn this block in four different ways and get four different pictures, unlike `Fill` or `Cross`. So, it definitely has an orientation. Its orientation is defined to be either 45, 135, 225, or 315 (the default is 45). We obtain that number by treating the bend as an "arrow", as you can see from Figure 1. We can then calculate how far counterclockwise from the horizontal this arrow is rotated – just like when you measured angles way back in geometry class. The `Bend` picture in Figure 1 "points" to the northeast, which is 45 degrees, so the orientation of that block is defined to be 45. If that block was rotated 90 degrees counterclockwise, it would be "pointing" to the northwest, and so its orientation would be defined to be 135. And so on. The two entrances are numbered by noticing whether they are to the right or the left of this imaginary dotted line that moves along the diagonal of the block, and this numbering scheme is clearly shown in Figure 1. `Bend` pictures can also be drawn on the screen in asterisk format – and in four different rotations, at that. The first one below is the one shown in the Figure 1 – the one with orientation 45. The order after that is 135, 225, and 315.

```
***     ***     *.*     *.*
..*     *..     *..     ..*
*.*     *.*     ***     ***
```

The `Tee` class has orientation defined to be 0, 90, 180, or 270, depending on which direction the "tee part" points in (see Figure 1). There are three entrances, numbered as shown in Figure 1. Plus, we can also do 3-by-3 asterisk drawings of `Tee` diagrams.

The `Straight` class has only two orientations – horizontal and vertical. We will call these 0 and 90. The first rotation we could do would move it from its current horizontal orientation, 90 degrees counterclockwise to vertical. The second rotation would move it 270 degrees counterclockwise to take it back to the original position. Whereas with `Tee` and `Bend`, you would need to rotate them four times to return them to their original orientations. (The reason we don't allow four 90-degree

rotations on `Straight` objects is so we don't run into the kind of problem where the 0 degree rotation and the 180-degree rotation would be exactly the same except for door numbers. We want our orientation possibilities to actually look different, and so for `Straight` objects, there are only two options.)

The first three classes are already written for you, and you can examine their code as you write the code for `Tee` and `Straight`. Much of it will be very similar.

The specification for the `Tee` class is as follows:

`Tee`

`Private Data Members`

As with the given code, your `Tee` class will need a 3-by-3 character array to hold its asterisk design. In addition, the class needs space for three entrances. Each entrance is either open (1) or closed (0) so it is best to use integers to represent the entrances so that you can record the status of entrances. See the other classes for help. Finally, you also need an orientation variable to hold the object orientation – either 0, 90, 180, or 270.

`Public Functions`

There are five public functions to write for `Tee`.

First, you will want a constructor, whose job is it to make sure that all entrances are initially open, the initial orientation is as shown in the above figure, and the internal array holds a correct asterisk picture of the current orientation of the object. You have not dealt with constructors a great deal yet, but you at least know what they are from lecture, and between that and the three classes of examples, you should be okay. Just follow the format of the given code.

The second and third functions are `OpenEntrance` and `CloseEntrance`. These functions work as in the give code – pass in the number of the entrance and that entrance is then either open or closed, depending which function you called. Note that it is perfectly reasonable to call `OpenEntrance` on an entrance that is already open, and vice-versa. Note also that you want to print out an appropriate error message if the index passed to these functions is greater than the number of entrances the block has.

The fourth function you need is `Rotate` – it takes no parameters, but it moves this object 90 degrees clockwise. So, if a `Tee` object had orientation 45, and you called `Rotate` on that object, it would then have orientation 135.

Finally, you need `Imprint`. `Imprint` will take an array, and will write this class's asterisk pattern into the array at locations indicated by the other parameters. See other classes for info; the function header for `Tee` will be identical to that of `Fill` or `Cross` or `Bend`, except that this one will be scoped to `Tee`.

`Straight` – you also need the class `Straight`, but the member functions are exactly the same except for the adjustments you make to take this tubing into account. For example, `Tee` has three entrances, but `Straight` only has two. That will result in certain differences in `Straight` in comparson to `Tee`. Also, `Straight` only has two possible rotation positions (as explained earlier) whereas `Tee` has four, so that will result in some differences as well. Plus, of course, the picture is different. :-) It is up to you to figure out exactly how `Straight`'s code will differ from `Tee`'s code and the code of the given classes. But, the number of member functions, and their names, and their parameters, and their ultimate purposes, are exactly the same.

5.2 Testing the new classes

The final actual code you need to write is the file `main.C`

Here, in order, is what you need to do in `main.C`

1. Dynamically allocate arrays of 10 elements of each of the five shape types. This will require declaring pointers to those types and then using `new` for the array allocation. We will not need all this space for *any* of the arrays – I will have you use at most 2 or 3 of the 10 cells in each array – but dynamically allocate arrays of size 10 anyway.

2. We also want a count of how many objects of each kind of shape we have used so far, so declare an integer variable corresponding to each shape class and set them all to 0. As we make use of objects in the arrays we just allocated, these counts will go up.

3. Declare a 3-by-3 array of `Indicator` pointers.

4. Create the following tube picture.

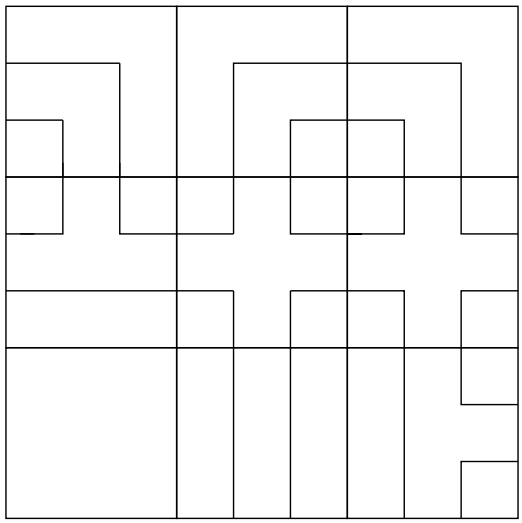

Figure 2: The five tube shapes in this MP

This is done by dynamically allocating `Indicator` objects, to be pointed to by the `Indicator pointers` in your 3-by-3 array. What do you pass into the `Indicator` constructor? Well, a character and an integer are what the constructor needs as parameters. (See the `Indicator` class.) What the `Indicator` objects are doing is storing the type and index information of the object you would like to put here. So, if you wanted, for example, the fifth `Tee` object, you would pass in `('t', 5)`, since 't' is the first letter of `Tee` and you wanted the fifth of the 10 `Tee` objects. Likewise, if you wanted the third `Cross` object, you would pass in `('c', 3)` to the `Indicator` constructor.

We want you to use the objects in the 5 dynamically allocated shape arrays from *low* index to *high* index. That is, you have 10 objects of some shape, and the first one you use should be at index 0. If you are supposed to use a second one, that one should be at index 1. If you are supposed to use a third one, that one should be at index 2. And so on.

5. Allocate a picture array that is a 9-by-9 array of characters. This corresponds to the 3-by-3 array of `Indicators` because each `Indicator` is the marker for an object of one of our shape classes. So, if you have a 3-by-3 array of Indicators, and each one refers to a class that can draw a 3-by-3 picture of itself, that results in a 3-by-3 grid of 3-by-3 grids, which is why we want a 9-by-9 grid for what we are about to do.

   What you are going to do here is traverse through the position array, i.e. the `Indicator` pointer array, and, at each position, access the Indicator object there, read its values, and use those values to determine what object you want out of the 50 objects in your five dynamic arrays. Then, once you have obtained this object, you want to pass values to its `Imprint` function. Specifically, you want pass it your picture array, and also the row and column number corresponding to this object's location in the `Indicator` pointer array.

   Huh? :-)

   For example, if you are at (2, 0) in your `Indicator` pointer array, access the `Indicator` object that yor array has a pointer to. Let's assume for a moment that the values you have in that `Indicator` are 'b' and '3'. That means the object that "belongs" here is the third `Bend` object. So, it is the third `Bend` object that you are now dealing with, because that is the object that the position (2, 0) in your position array was referring you to. Now, invoke that object's `Imprint` function, and pass it the picture array and the values (2, 0).

   What this will do is write the 3 by 3 asterisk pattern for your `Bend` object (or whichever object it was) into the appropriate spot in the 9 by 9 picture array. Trace through the code and see for yourself!

6. Once you have run all nine `Imprint` calls, print out how many of each object are in your diagram. (see `test.1.std`)

7. Print the picture array. (Hint: we've done a lot of this work for you already. Look at the files.) When you've finished this you are done with your output.

8. Don't forget to delete your dynamically allocated memory!

Once you finish this, you are done! Now, go ahead and `make`, and proceed with the MP, correcting any syntax errors that come up.

6 The overall goal

This MP is as much reading as it is coding – as stated earlier, if you have a decent knowledge of the given code, and understand how it all works, then you should find that writing the two shape classes is not that difficult. The `main.C` file is a little bit trickier. Look over lecture and section notes on dynamic allocation, and read through a C++ text as well. However, it *is* possible to finish the assignment and yet skip out on becoming thoroughly familar with the code, and it *is* possible to complete the assignment without running through the tutorials in the first half of the MP. This is not a good idea, however, for two reasons:

1. First and foremost, you will have difficulties later. Usually, we are not going to give you a nice chunk of time and a simple assignment. So, since you have extra time, *now* is when you should take advantage of that time to learn the software tools and review C++. If you wait until the second or third MP, you will have less time and will find things to be much harder going.

2. In addition, if you come into office hours for help, we will assume you are familiar with these tools, at least to the extent that the tutorials and MP have gone over them. Certainly, if later on you approach us with a question and it turns out you have a small problem with your `Makefile` due to a mistake coding it, that's understandible. However, if you cannot answer a simple question about the Makefile, or if we ask you to pull up the debugger and you have no idea how, we *do* reserve the right to not help you in office hours until you have gone through the tutorials and at least become *somewhat* familiar with the basics of these tools. (Certainly, though, if you have a question specifically about the tools, you are more than welcome to ask!)

Ultimately, one of the things we'd like you to get out of this course is *self-sufficiency*. You will be *far* better prepared for your future courses if you try things on your own first, and spend some time trying to track down your errors before coming to us for help. We are here to help you when you get confused or stuck, but you will not get everything out of this class that you can if you run to us *first*, without first attempting to solve your problems yourself. So, one of the reasons we provide you with these tutorials is so that you can learn the tools needed to help yourself. When you hit a wall, and cannot figure out a specific problem, then you should approach us with your question – but only then, after you have spent some time trying to solve it and are honestly stuck. If you want to learn the most you can from this course, that is the best way to go about your work.

7 Handing in

To handin your MP1 code, use the following command:

```
handin cs225 mp1 tee.h tee.C straight.h straight.C main.C
```

Time Recording Log

Student ______________________________ Date ______________

Instructor ______________________________ Class ______________

Date	Start	Stop	Interruption Time	Delta Time	Activity	Comments	C	Units