YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

CPSC 223b: Data Structures and Programming Techniques *Handout #11*
Professor M. J. Fischer *February 18, 2008*

Modules and Separate Compilation

1 Modules

A C program can be broken into several files that are compiled separately and then linked together to form a runable program. A group of related files is called a *module*. A well-designed module contains related declarations and definitions and has a well-defined meaning.

There are several reasons for structuring a program into modules.

1. Smaller files are easier for the programmer to deal with. When modifying the program, a programmer is less likely to accidentally modify an unrelated part of the program.

2. Modules provide a semantic structure to which the program can be related. This makes the program easier to understand. Instead of trying to keep the entire program in mind at once, it suffices to check that each module meets its specifications under the assumption that all other modules also meet their specifications. This breaks down the problem of understanding a complex program into the independent problems of understanding each module individually. Of course, for this to work, one should carefully think through and document the specifications for each module.

3. Modules make the program easier to debug. The same semantic structure that aids in understanding a program can also be used in debugging. Rather than only testing that the entire program works, one can and should test each individual module to ensure that it meets its specifications. Where feasible, modules should be tested independently of the rest of the program. This allows much more thorough testing of the module since test cases can be constructed to exercise specific parts of the module that may be invoked only rarely during regular use.

4. Modules can be compiled separately. When developing a project, modules that have not been changed do not have to be recompiled each time the program is modified. While compilation time is no longer an issue with programs the size of student exercises, it is still a major factor when working with large applications and systems that may be comprised of hundreds or even thousands of modules.

5. Modules with well-defined interfaces can be reused in other programs with no change. (All of the functions in the standard C library fall in this class.)

Roughly speaking, a module consists of two parts: An *interface*, which contains the information needed by other programs in order to use the services provided by the module, and an *implementation*, which is the code that actually defines the functions of the module. C provides two mechanisms for supporting modules: the `#include` statement and the ability to link together several separately compiled code (`.c`) files. `#include` allows an interface, represented by a header (`.h`) file, to be be accessed at compile time by other modules that use it. Separate compilation allows an implementation, represented by a separately-compiled code (`.c`) file, to be incorporated into the complete executable program.

2 #include and the C Preprocessor

#include is a directive to the C compiler to read and process a header file just as if it had been copied into the source code being compiled. In fact, all of the # directives (such as #define, which you have already seen, and #ifdef which we have not yet talked about) are processed by the C preprocessor, which makes a preliminary pass through your program before the actual compilation begins. During that pass, #include gets replaced by the contents of the included file, symbols defined by #define get replaced by their definitions, and other # directives get interpreted.

The reason for using #include is so that you don't have to know the details of what is being included, nor do you have to change your program if the header file is subsequently updated. All that is necessary is to recompile the program and the updated file will be included and used.

There is nothing in C to prevent included files from containing function definitions (as opposed to prototypes), but they are rarely used that way. Rather, header files are usually used to implement interfaces, in which case they will contain only declarations and other information for the compiler such as #define directives, typedef type declarations, function prototypes, and global variable declarations in those rare instances where use of global variables is warranted. (For an example of a global variable used by the standard libraries, see the errno man page.) Header files are also permitted to #include other header files, but one must take care to avoid circular includes and having the same file included multiple times. Techniques for handling such cases are beyond the scope of this writeup.

There are actually two forms of #include, depending on whether the file name is enclosed in angle brackets or in double quotes. #include <stdio.h> says to look for stdio.h in the system directories where standard header files are stored.[1] #include "myheader.h" looks for a file named myheader.h, searching personal directories first. The -I switch to gcc can be used to specify additional directories to be searched. So the rule is, use #include <...> for system headers, #include "..." for user-defined headers.

3 Separate Compilation

Building a runnable program in C occurs in two separate phases. The *compilation* phase reads C source code and translates it into a form called *object code* that the processor can execute directly. The translation process discards information in the source file such as comments and symbols that are not needed by the running code, preserving only the semantic content of the code (i.e., the machine code should carry out the computations specified by the source code).[2] The object code is written to a file with the .o suffix.

The *linking* phase puts all of the pieces of a program together to form a runnable command. Most programs, even simple ones, are not self-contained but refer to functions defined in the system libraries. Similarly, most modules will refer to functions defined in other modules and in the system libraries.

The gcc command can be called in different ways to do just the compilation phase, just the linking phase, or both. The -c flag says to just do compilation. Without the -c flag, if all file arguments to the command are .o files, then it just does the linking. Otherwise, it does both, compiling any .c files it finds on the command line and then linking together the resulting object

[1]Strictly speaking, standard headers are not even required to be actual files, but their contents must somehow become "known" to the compiler in response to the #include statement. On the Zoo, stdio.h is found in the directory /usr/include.

[2]With the -g flag, the compiler preserves symbols for use by the debugger.

files with any other `.o` files on the command line and with the system libraries.

```
1.   gcc -c -std=c99 -pedantic -Wall -g -O1 listsort.c
2.   gcc -c -std=c99 -pedantic -Wall -g -O1 list.c
3.   gcc -o listsort listsort.o list.o
```

Figure 1: Three steps in building `listsort`.

For example, if `listsort.c` and `list.c` are modules comprising the command `listsort`, then three steps are necessary to produce the executable file `listsort` are shown in Figure 1. Line 1 generates `listsort.o`. Line 2 generates `list.o`. Line 3 links `listsort.o` and `list.o` together along with any system library routines used by the code such as `fscanf()`. The standard libraries are searched by default. Additional libraries can be specified using the `-l` flag. For example, `-lm` causes the math library to be searched.

The flags `-std=c99 -pedantic -Wall -g -O1` in lines 1 and 2 are the options to `gcc` that should be used for this course. `-std=c99` instructs the compiler to (mostly) follow the C99 standard. `-pedantic` disables non-conforming extensions to the standard. `-Wall` causes all warnings to be printed. `-g` tells the compiler to include the symbol table in the `.o` file (useful with the `gdb` debugger). `-O1` ("oh one") enables first-level code optimization which, as a byproduct, enables some helpful warnings to be issued for conditions that otherwise are not detected.

The three steps could be combined in a single command as shown in Figure 2. This would generate the executable `listsort` but would omit writing the two `.o` files to disk.

```
gcc -o listsort -c -std=c99 -pedantic -Wall -g -O1 listsort.c list.c
```

Figure 2: Combining the three steps of Figure 1.

4 Makefiles

The `make` facility is one way to automate the build process. The `Makefile` describes the structure of your code. Then when you type `make`, it looks at your files and builds any of the derived files that are missing or out of date.

In the above example, the derived files are the three files `listsort.o`, `list.o`, and `listsort` produced by the three steps of Figure 1. A file is out of date if its timestamp is older than one of the files upon which it depends. In this example, `listsort.o` depends on `listsort.c`, `list.o` depends on `list.c`, and `listsort` depends on both `listsort.o` and `list.o`.

Dependencies are applied recursively. Before testing the timestamps of `listsort.o` and `list.o` to determine if `listsort` needs rebuilding, `listsort.o` and `list.o` are checked to see if they are up to date by looking at the files *they* depend on, and they are rebuilt if necessary. Of course, once any of them has been rebuilt, the newly-built version will have a newer timestamp than `listsort`, causing `listsort` to be rebuilt as well.

A `.o` file might depend on other files in addition to the corresponding `.c` file. For example, if `list.c` contains the line `#include "list.h"`, then the contents of `list.h` may affect the resulting `.o` file, so if `list.h` changes, then `list.c` must be recompiled to produce an up-to-date version of `list.o`. Because `make` only knows what you tell it in the `Makefile`, you would need to give `list.h` as an additional dependency for `list.c` (and any other file that includes it, such as `listsort.c`) in `Makefile`.

```
CFLAGS = -std=c99 -pedantic -Wall -g -O1
listcode:  listcode.o list.o
	$(CC) -o listcode listcode.o list.o
listcode.o:  listcode.c list.h
list.o:  list.c list.h
```

Figure 3: A minimal `Makefile`.

A minimal `Makefile` for doing what I just described is shown in Figure 3. The first line defines the flags we want to use when compiling `.c` files. The next line shows the files that `listcode` depends on, and the third line shows how to build `listcode` from those two dependencies. The last two lines show the dependencies for `listcode.o` and `list.o`. We don't have to describe how to build a `.o` file from a `.c` file because `make` already knows how to do that. However, by setting the macro `CFLAGS`, we are giving `make` additional flags to pass to the compiler when building `.o` files.

5 Writing Your Own Modules

The two parts of a module are represented by two different files in C. The interface is given by a `.h` *header* file, and the implementation is given by a `.c` *code* file. Here's what goes in each file, and why.

The implementation file must contain everything that a calling module needs to know in order to properly compile a call to one of the functions defined in the current module. For example, if the control break module contains a function with prototype

```
void dotrans( transaction* tr, double* st );
```

then any program that calls `dotrans()` needs that information. But that isn't enough. It also needs the definition of the user-defined type `transaction`.

Another program that wants to call `dotrans()` could copy the following lines to the top of the source file:

```
typedef
struct {
   char date[10];
   char store[30];
   double amount;
} transaction;
void dotrans( transaction* tr, double* st );
```

However, this would not be a good idea because violates the principle of locality which says that related things belong together. It also violates the principle that dictates against duplication of code. The typedef for `transaction` is part of the definition of the module and belongs in the files associated with the module, not in the client program.

These problems are both solved by putting the necessary type declarations and prototypes together into a header file, `ctlbreak.h`. The calling program now need only `#include "ctlbreak.h"`. When the calling program is compiled, the compiler will read the header file and process the declarations contained therein, enabling it to correctly compile the call to `dotrans()`.

Now, the actual definition of `dotrans()` needs to placed somewhere and compiled so that it is available to be linked in with the executable. This goes into the implementation file, `ctlbreak.c`. Why doesn't the code go into `ctlbreak.h`, along with the type definitions and prototypes? The reason is that the definitions and prototypes are needed by every program that uses the control break module, whereas the code gets compiled only once and then shared by all programs that need it. In fact, if it were placed in the header file, it would get compiled into the `.o` file of every module that used it. Later, when those files are linked together, the linker would find multiple copies of the same function and not know which one to use, resulting in a fatal linking error. So it is important that the defining code for each function appear in only one `.c` file, and that it not be part of any `.h` file.

The implementation file also needs the definitions contained in the interface file, so it also needs to `#include` the interface. In our example, the implementation file `ctlbreak.c` might look in part as follows:

```
#include "ctlbreak.h"

void dotrans( transaction* tr, double* st )
{
  printf( "%9s  %-29s  %8.2f\n", tr->date, tr->store, tr->amount);
  *st += tr->amount;
}
```

Of course, in real life the notion which this module is trying to model would probably have more than one function, so `ctlbreak.c` would contain a collection of function definitions, not just one.

6 Testing a Module

Testing is an important part of the code development process that is often shortchanged to the detriment of the final product. Testing takes time and effort. When working under a deadline, it's easy to imagine that the project will be finished sooner if the testing is skipped. It won't be! The goal with debugging is to find and eliminate bugs as efficiently as possible. The bigger the piece of code and the more bugs are present, the harder it is to figure out what is going on and fix the bugs. Bugs that are caught by the compiler are the easiest to fix. That's why warnings shouldn't be ignored—they often indicate errors in program logic that will be difficult to track down if they are ignored. Bugs that only appear during testing are harder to track down, but if the code is tested a piece at a time, the bugs that do appear are relatively easy to localize. Only after each module has been thoroughly tested does one want to go on to test the complete program. At that point, the bugs that are likely to appear will be ones that result from unanticipated interactions of the modules with each other, but at least then you'll have some confidence that each individual module is behaving as expected.

To test a module will generally require that additional testing code be written. For example, to test the `dotrans()` function that we have been discussing, you might write a little test program that creates a transaction, initializes it with data, and calls `dotrans()`. The testing program would then check that `st` was incremented properly, and you would verify that the printed output was correct.

Another strategy for testing involves *instrumenting* your code. That means going into the implementation file and adding statements to check and/or print out what is going on in the code. I use the convention of starting each line of debugging printouts with ">>>" as in

```
printf(">>> function dotrans() entered\n");
```

That makes it easy to distinguish debugging output from real output during testing. It also makes the debugging statements easy to locate and remove later. Some people would argue that the debugging statements should remain in the code and be enabled or disabled conditionally, e.g.,

```
if (DEBUG) printf(">>> function dotrans() entered\n");
```

where `DEBUG` can be set to 0 or 1 using `#define`. The downside of leaving them in is that they tend to clutter up the code and make it harder to read. Also, unless they are carefully designed, they probably will be of little use later on, especially if they were inserted in the first place for the purpose of tracking down a particular bug. This, as with many other aspects of C programming, requires judgment that can only be developed with experience.

Finally, other testing and debugging tools can sometimes be quite useful. `valgrind` will run your program in a mode that checks for memory management problems such as invalid memory accesses, memory leaks, and failure to free memory. The Gnu debugger `gdb` will run your program a few steps at a time or up to the next preset breakpoint, where it will then pause. At that point, you can examine and/or modify the contents of program variables to try to figure out what is going on. Where `gdb` is particularly useful is in the case of a segmentation fault, for the first thing you want to know in that case is where you were in the program when the crash happened. After the crash, the `backtrace` command to `gdb` will print out the chain of function calls that led up to the crash.