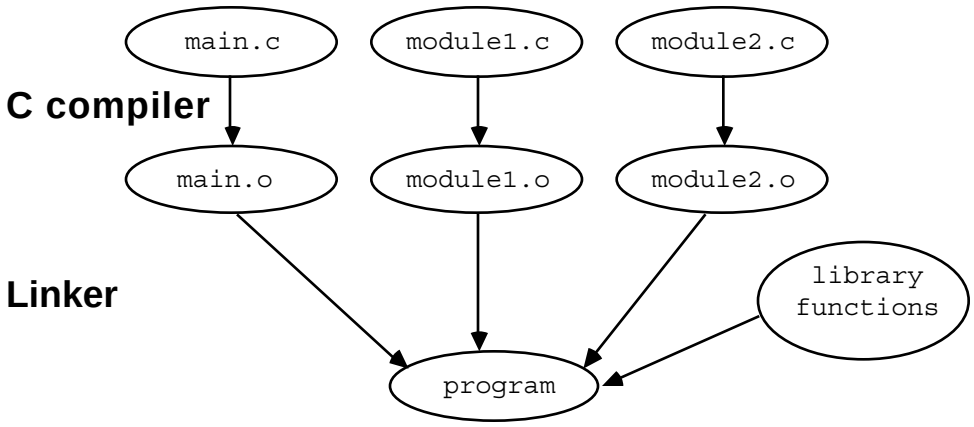
also bring in library object files that contain the definitions of library functions like `printf()` and `malloc()`. The overall process looks like this...



Section 1 — gcc

The following discussion is about the gcc compiler, a product of the open-source GNU project (www.gnu.org). Using gcc has several advantages— it tends to be pretty up-to-date and reliable, it's available on a variety of platforms, and of course it's free and open-source. Gcc can compile C, C++, and objective-C. Gcc is actually both a compiler and a linker. For simple problems, a single call to gcc will perform the entire compile-link operation. For example, for small projects you might use a command like the following which compiles and links together three .c files to create an executable named "program".

```
gcc main.c module1.c module2.c -o program
```

The above line equivalently could be re-written to separate out the three compilation steps of the .c files followed by one link step to build the program.

```
gcc -c main.c                                ## Each of these compiles a .c
gcc -c module1.c
gcc -c module2.c
gcc main.o module1.o module2.o -o program       ## This line links the .o's
                                                ## to build the program
```

The general form for invoking gcc is...

```
gcc options files
```

where *`options`* is a list of command flags that control how the compiler works, and *`files`* is a list of files that gcc reads or writes depending on the options

Command-line options

Like most Unix programs, gcc supports many command-line options to control its operation. They are all documented in its man page. We can safely ignore most of these options, and concentrate on the most commonly used ones: `-c, -o, -g, -Wall, -I, -L, and -l`.

`-c files` Direct gcc to compile the source files into an object files without going through the linking stage. Makefiles (below) use this option to compile files one at a time.

`-o file` Specifies that gcc's output should be named *file*. If this option is not specified, then the default name used depends on the context...(a) if compiling a source .c file, the output object file will be named with the same name but with a .o extension. Alternately, (b) if linking to create an executable, the output file will be named `a.out`. Most often, the -o option is used to specify the output filename when linking an executable, while for compiling, people just let the default .c/.o naming take over.

It's a memorable error if your -o option gets switched around in the command line so it accidentally comes before a source file like "`...-o foo.c program`" -- this can overwrite your source file -- bye bye source file!

`-g` Directs the compiler to include extra debugging information in its output. We recommend that you always compile your source with this option set, since we encourage you to gain proficiency using the debugger such as gdb (below).

Note -- the debugging information generated is for gdb, and could possibly cause problems with other debuggers such as dbx.

`-Wall` Give warnings about possible errors in the source code. The issues noticed by -Wall are not errors exactly, they are constructs that the compiler believes may be errors. We highly recommend that you compile your code with -Wall. Finding bugs at compile time is soooo much easier than run time. the -Wall option can feel like a nag, but it's worth it. If a student comes to me with an assignment that does not work, and it produces -Wall warnings, then maybe 30% of the time, the warnings were a clue towards the problem. 30% may not sound like that much, but you have to appreciate that it's **free** debugging.

Sometimes -Wall warnings are not actually problems. The code is ok, and the compiler just needs to be convinced. Don't ignore the warning. Fix up the source code so the warning goes away. Getting used to compiles that produce "a few warnings" is a very bad habit.

Here's an example bit of code you could use to assign and test a flag variable in one step...

```
int flag;

if (flag = IsPrime(13)) {
  ...
}
```

The compiler will give a warning about a possibly unintended assignment, although in this case the assignment is correct. This warning would catch the common bug where you meant to type == but typed = instead. To get rid of the warning, re-write the code to make the test explicit...

```
int flag;

if ((flag = IsPrime(13)) != 0) {
  ...
}
```

This gets rid of the warning, and the generated code will be the same as before. Alternately, you can enclose the entire test in another set of parentheses to indicate your intentions. This is a small price to pay to get -Wall to find some of your bugs for you.

`-I`*`dir`* Adds the directory *dir* to the list of directories searched for #include files. The compiler will search several standard directories automatically. Use this option to add a directory for the compiler to search. There is no space between the "-I" and the directory name. If the compile fails because it cannot find a #include file, you need a -I to fix it.

Extra: Here's how to use the unix "find" command to find your #include file. This example searches the /usr/include directory for all the include files with the pattern "inet" in them...

```
nick% find /usr/include -name '*inet*'
/usr/include/arpa/inet.h
/usr/include/netinet
/usr/include/netinet6
```

`-l`*`mylib`* (lower case 'L') Search the library named *mylib* for unresolved symbols (functions, global variables) when linking. The actual name of the file will be lib*mylib*.a, and must be found in either the default locations for libraries or in a directory added with the `-L` flag (below).

The position of the `-l` flag in the option list is important because the linker will not go back to previously examined libraries to look for unresolved symbols. For example, if you are using a library that requires the math library it must appear before the math library on the command line otherwise a link error will be reported. Again, there is no space between the option flag and the library file name, and that's a lower case 'L', not the digit '1'. If your link step fails because a symbol cannot be found, you need a -l to add the appropriate library, or somehow you are compiling with the wrong name for the function or global variable.

`-L`*`dir`* Adds the directory *dir* to the list of directories searched for library files specified by the `-l` flag. Here too, there is no space between the option flag and the library directory name. If the link step fails because a library file cannot be found, you need a -L, or the library file name is wrong.

you need to recompile from scratch. The `TAGS` rule creates a tag file that most Unix editors can use to search for symbol definitions.

Compiling in Emacs

Emacs has built-in support for the compile process. To compile your code from emacs, type `M-x compile`. You will be prompted for a compile command. If you have a makefile, just type `make` and hit return. The makefile will be read and the appropriate commands executed. The emacs buffer will split at this point, and compile errors will be brought up in the newly created buffer. In order to go to the line where a compile error occurred, place the cursor on the line which contains the error message and hit `^c-^c`. This will jump the cursor to the line in your code where the error occurred ("cc" is the historical name for the C compiler).

Section 3 — gdb

You may run into a bug or two in your programs. There are many techniques for finding bugs, but a good debugger can make the job a lot easier. In most programs of any significant size, it is not possible to track down all of the bugs in a program just by staring at the source — you need to see clues in the runtime behavior of the program to find the bug. It's worth investing time to learn to use debuggers well.

GDB

We recommend the GNU debugger gdb, since it basically stomps on dbx in every possible area and works nicely with the gcc compiler. Other nice debugging environments include ups and CodeCenter, but these are not as universally available as gdb, and in the case of CodeCenter not as cheaply. While gdb does not have a flashy graphical interface as do the others, it is a powerful tool that provides the knowledgeable programmer with all of the information they could possibly want and then some.

This section does not come anywhere close to describing all of the features of gdb, but will hit on the high points. There is on-line help for gdb which can be seen by using the `help` command from within gdb. If you want more information try `xinfo` if you are logged onto the console of a machine with an X display or use the info-browser mode from within emacs.

Starting the debugger

As with make there are two different ways of invoking gdb. To start the debugger from the shell just type...

`gdb` *`program`*

where *`program`* is the name of the target executable that you want to debug. If you do not specify a target then gdb will start without a target and you will need to specify one later before you can do anything useful.

As an alternative, from within emacs you can use the command `[Esc]-x gdb` which will then prompt you for the name of the executable file. You cannot start an inferior gdb session from within emacs without specifying a target. The emacs window will then split between the gdb buffer and a separate buffer showing the current source line.

Running the debugger

Once started, the debugger will load your application and its symbol table (which contains useful information about variable names, source code files, etc.). This symbol

table is the map produced by the -g compiler option that the debugger reads as it is running your program.

The debugger is an interactive program. Once started, it will prompt you for commands. The most common commands in the debugger are: setting breakpoints, single stepping, continuing after a breakpoint, and examining the values of variables.

Running the Program

`run`	Reset the program, run (or rerun) from the beginning. You can supply command-line arguments the same way you can supply command-line arguments to your executable from the shell.
`step`	Run next line of source and return to debugger. If a subroutine call is encountered, follow into that subroutine.
`step` *`count`*	Run *count* lines of source.
`next`	Similar to step, but doesn't step into subroutines.
`finish`	Run until the current function/method returns.
`return`	Make selected stack frame return to its caller.
`jump` *`address`*	Continue program at specified line or address.

When a target executable is first selected (usually on startup) the current source file is set to the file with the main function in it, and the current source line is the first executable line of the this function.

As you run your program, it will always be executing some line of code in some source file. When you pause the program (when the flow of control hits a “breakpoint” of by typing Control-C to interrupt), the “current target file” is the source code file in which the program was executing when you paused it. Likewise, the “current source line” is the line of code in which the program was executing when you paused it.

Breakpoints

You can use breakpoints to pause your program at a certain point. Each breakpoint is assigned an identifying number when you create it, and so that you can later refer to that breakpoint should you need to manipulate it.

A breakpoint is set by using the command `break` specifying the location of the code where you want the program to be stopped. This location can be specified in several ways, such as with the file name and either a line number or a function name within that file (a line needs to be a line of actual source code — comments and whitespace don't count). If the file name is not specified the file is assumed to be the current target file, and if no arguments are passed to `break` then the current source line will be the breakpoint. `gdb` provides the following commands to manipulate breakpoints:

`info break`	Prints a list of all breakpoints with numbers and status.

`break` *`function`*	Place a breakpoint at start of the specified function
`break` *`linenumber`*	Prints a breakpoint at line, relative to current source file.
`break` *`filename:linenumber`*	Place a breakpoint at the specified line within the specified source file.

You can also specify an *if* clause to create a conditional breakpoint:

`break` *`fn`* `if` *`expression`*	Stop at the breakpoint, only if *expression* evaluates to true. Expression is any valid C expression, evaluated within current stack frame when hitting the breakpoint.
`disable` *`breaknum`* `enable` *`breaknum`*	Disable/enable breakpoint identified by *breaknum*
`delete` *`breaknum`*	Delete the breakpoint identified by *breaknum*
`commands` *`breaknum`*	Specify commands to be executed when *breaknum* is reached. The commands can be any list of C statements or gdb commands. This can be useful to fix code on-the-fly in the debugger without re-compiling (Woo Hoo!).
`cont`	Continue a program that has been stopped.

For example, the commands...

```
break binky.c:120
break DoGoofyStuff
```

set a breakpoint on line 120 of the file binky.c and another on the first line of the function `DoGoofyStuff`. When control reaches these locations, the program will stop and give you a chance to look around in the debugger.

Gdb (and most other debuggers) provides mechanisms to determine the current state of the program and how it got there. The things that we are usually interested in are (a) where are we in the program? and (b) what are the values of the variables around us?

Examining the stack

To answer question (a) use the `backtrace` command to examine the run-time stack. The run-time stack is like a trail of breadcrumbs in a program; each time a function call is made, a crumb is dropped (an run-time stack frame is pushed). When a return from a function occurs, the corresponding stack frame is popped and discarded. These stack frames contain valuable information about the sequence of callers which brought us to the current line, and what the parameters were for each call.

Gdb assigns numbers to stack frames counting from zero for the innermost (currently executing) frame. At any time gdb identifies one frame as the “selected” frame. Variable lookups are done with respect to the selected frame. When the program being debugged stops (at a breakpoint), gdb selects the innermost frame. The commands below can be used to select other frames by number or address.

`backtrace`	Show stack frames, useful to find the calling sequence that produced a crash.
`frame` *framenumber*	Start examining the frame with *framenumber*. This does not change the execution context, but allows to examine variables for a different frame.
`down`	Select and print stack frame called by this one. (The metaphor here is that the stack grows down with each function call.)
`up`	Select and print stack frame that called this one.
`info args`	Show the argument variables of current stack frame.
`info locals`	Show the local variables of current stack frame.

Examining source files

Another way to find our current location in the program and other useful information is to examine the relevant source files. `gdb` provides the following commands:

`list` *linenum*	Print ten lines centered around *linenum* in current source file.
`list` *function*	Print ten lines centered around beginning of *function (*or *method).*
`list`	Print ten more lines.

The `list` command will show the source lines with the current source line centered in the range. (Using `gdb` from within `emacs` makes these command obsolete since it does all of the current source stuff for you.)

Examining data

To answeer the question (b) “what are the values of the variables around us?” use the following commands...

`print` *expression*	Print value of *expression.* Expression is any valid C expression, can include function calls and arithmetic expressions, all evaluated within current stack frame.
`set` *variable* `=` *expression*	Assign value of *variable* to *expression*. You can set any variable in the current scope. Variables which begin with $ can be used as temporary variables local to gdb.
`display` *expression*	Print value of *expression* each time the program stops. This can be useful to watch the change in a variable as you step through code.
`undisplay`	Cancels previous display requests.

In gdb, there are two different ways of displaying the value of a variable: a snapshot of the variable's current value and a persistent display for the entire life of the variable. The `print` command will print the current value of a variable, and the `display` command will make the debugger print the variable's value on every step for as long as the variable exists. The desired variable is specified by using C syntax. For example...

```
print x.y[3]
```

will print the value of the fourth element of the array field named `y` of a structure variable named `x`. The variables that are accessible are those of the currently selected function's activation frame, plus all those whose scope is global or static to the current target file. Both the `print` and `display` functions can be used to evaluate arbitrarily complicated expressions, even those containing, function calls, but be warned that if a function has side-effects a variety of unpleasant and unexpected situations can arise.

Shortcuts

Finally, there are some things that make using gdb a bit simpler. All of the commands have short-cuts so that you don't have to type the whole command name every time you want to do something simple. A command short-cut is specified by typing just enough of the command name so that it unambiguously refers to a command, or for the special commands `break`, `delete`, `run`, `continue`, `step`, `next` and `print` you need only use the first letter. Additionally, the last command you entered can be repeated by just hitting the return key again. This is really useful for single stepping for a range while watching variables change.

Miscellaneous

`editmode` *`mode`*	Set editmode for gdb command line. Supported values for *mode* are emacs, vi, dumb.
`shell` *`command`*	Execute the rest of the line as a shell command.
`history`	Print command history.

Debugging Strategies

Some people avoid using debuggers because they don't want to learn another tool. This is a mistake. Invest the time to learn to use a debugger and all its features — it will make you much more productive in tracking down problems.

Sometimes bugs result in program crashes (a.k.a. "core dumps", "register dumps", etc.) that bring your program to a halt with a message like "Segmentation Violation" or the like. If your program has such a crash, the debugger will intercept the signal sent by the processor that indicates the error it found, and allow you to examine the state program. Thus with almost no extra effort, the debugger can show you the state of the program at the moment of the crash.

Often, a bug does not crash explicitly, but instead produces symptoms of internal problems. In such a case, one technique is to put a breakpoint where the program is misbehaving, and then look up the call stack to get some insight about the data and control flow path that led to the bad state. Another technique is to set a breakpoint at some point before the problems start and step forward towards the problems, examining the state of the program along the way.