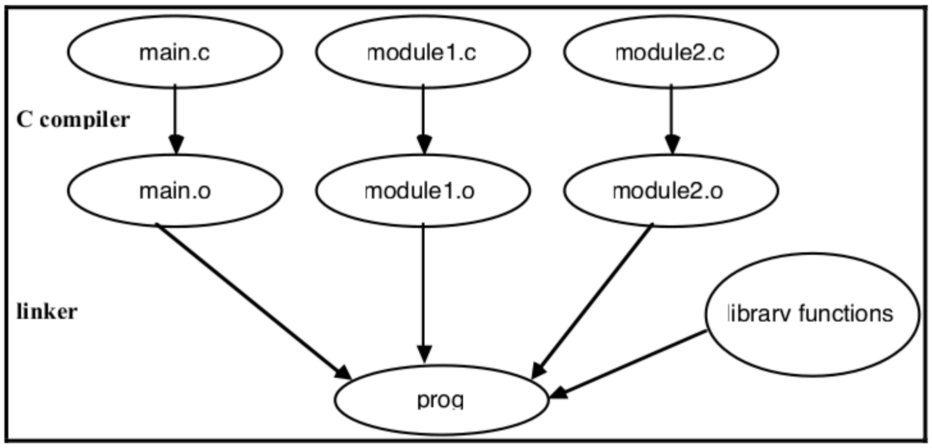
The Compilation Process

Before going into detail about the actual tools themselves, it is useful to review what happens during the construction of an executable program. There are actually two phases in the process, *compilation* and *linking*. The individual source files must first be *compiled* into object modules. These object modules contain a system dependent, relocatable representation of the program as described in the source file. The individual object modules are then *linked* together to produce a single executable file which the system loader can use when the program is actually invoked. (This process is illustrated by the diagram on the next page.) These phases are often combined with the `gcc` command, but it is quite useful to separate them when using `make`.
For example, the command: `gcc -o prog main.c module1.c module2.c`
can be broken down into the four steps of:

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



Compiler/Linker

Although there are a lot of different compilers out there, we “guarantee” that all of the problems can be solved using the GNU C Compiler, `gcc` to its friends. However, because C++ requires additional libraries, you should use `g++`, which is just a wrap-around of `gcc`, to compile C++ programs. Using `g++` has several advantages, it is pretty much ANSI compliant, available on a variety of different platforms and, more importantly for you, it works pretty reliably. The current version of `g++` installed on the L&IR machines is 2.8.1, and directly compiles C, C++, and Objective-C.

Running g++

Even though it is called a compiler, `g++` is used as both a compiler and linker. The general form for invoking `g++` is :

```
g++ <option flags> <file list>
```

where `<option flags>` is a list of command flags that control how the compiler works, and `<file list>` is a list of files, source or object, that `g++` is being directed to process. It is not, however, commonly invoked directly from the command line, that is what makefiles are for. If `g++` is unable to process the files correctly it will print error messages on standard error. Some of these error messages, however, will be caused by `g++` trying to recover from a previous error so it is best to try to tackle the errors in order.

Command-line options

Like almost all UNIX programs `g++` has myriad options that control almost every aspect of its actions. However, most of these options deal with system dependent features and do not concern us. The most useful option flags for us are: `-c, -o, -g, -Wall, -I, -L, and -l`.

`-c` Requests that `g++` compile the specific source file directly into an object file without going through the linking stage. This is important for compiling only those files that have changed rather than the whole project.

`-o` *`file`* Specifies that you want the compiler's output to be named *file*. If this option is not specified, the default is to create a file '`a.out`' if you are linking object files into an executable. If you are compiling a source file (with suffix .c for files written in C) into an object file, the default name for the object file is simply the original name with the '.c' replaced with '.o'. This option is generally only used for creating an application with a specific name (during linking), rather than for making the names of object files differ from the default *source-filename*.o.

`-g` Directs the compiler to produce debugging information. We recommend that you always compile your source with this option set, since we encourage you to gain proficiency using the debugger (we recommend `gdb`).

Note – The debugging information generated is for `gdb`, and could possibly cause problems with `dbx`. This is because there is typically more information stored for `gdb` that `dbx` will barf on. Additionally, on some systems, some `MIPS` based machines for example, this information cannot encode full symbol information and some debugger features may be unavailable.

`-Wall` Give warnings about a lot of syntactically correct but dubious constructs. Think of this option as being a way to do a simple form of style checking. Again, we highly recommend that you compile your code with this option set.

Most of the time the constructs that are flagged are actually incorrect usages, but there are occasionally instances where they are what you really want. Instead of simply ignoring these warnings there are simple workarounds for almost all of the warnings if you insist on doing things this way.

This sort of contrived snippet is a commonly used construct in C to set and test a variable in as few lines as possible :

```
int flag;

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

The compiler will give a warning about a possibly unintended assignment. This is because it is more common to have a boolean test in the `if` clause using the equality operator == rather than to take advantage of the return value of the assignment operator. This snippet could better be written as :

```
int flag;

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

so that the test for the `0` value is made explicit. The code generated will be the same, and it will make us and the compiler happy at the same time. Alternately, you can enclose the entire test in another set of parentheses to indicate your intentions.

`-I`*`dir`* Adds the directory *dir* to the list of directories searched for include files. This will be important for any additional files that we give you. There are a variety of standard directories that will be searched by the compiler by default, for standard library and system header files, but since we do not have root access we cannot just add our files to these locations.

There is no space between the option flag and the directory name.

`-l`*`lib`* Search the library named *lib* for unresolved names when linking. The actual name of the file will be lib*lib*.a, and must be found in either the default locations for libraries or in a directory added with the `-L` flag.

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 names. 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.

`-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.