

# Programming in C and Pascal Under Unix

Rutgers University  
LCSR Computing Facility  
21 August 1996

---

There are a number of different programming languages available on our systems. The ones most commonly used in coursework are C, Fortran, Pascal, Lisp, and Prolog. This document talks most specifically about C and Pascal. However most of what it says applies also to the other languages that use compilers, such as Fortran. Lisp and Prolog are very different, and require their own documentation.

## 1. A Roadmap for Preparing and Debugging a Program

There are three basic steps to preparing and running a program:

- *Preparing the source files, using an editor.*
- *Compiling and building the program. This step reads the source files, and turns them into an executable program.*
- *Testing and debugging.*

We recommend using Emacs for all of these steps. While Emacs is basically an editor, it has special facilities to control compilers and debuggers. When you compile under the control of Emacs, Emacs will look at the error messages, and let you go directly to the appropriate place in the source file. When you use a debugger under the control of Emacs, Emacs will watch what is happening and point you to the appropriate line in the source file.

If you have used integrated programming development tools on a micro, you'll find that the Emacs environment has similar facilities, but it is not quite as automated. You must tell Emacs when you want to compile the program, and when you want to restart the debugger. Fortunately these things are easy to.

While this document is primarily oriented to the use of Emacs, it will also tell you the commands to type if you want to do things yourself.

### 1.1. What you need to know about Emacs

If you are using Emacs, we assume that you have gone through teach-emacs, and understand normal conventions for describing emacs commands, e.g. things like "m-x gdb" (which means to hit the ESC key, then "x", and then type "gdb"). As in most Emacs documentation, m-x will be used to indicate that you type ESC and then x, and c-x will be used to indicate Control-x.

In the past we have recommended a special package called "C Menus". This added some menus to make compiling and debugging easier. Unfortunately it has proven difficult to maintain this package as Emacs changes. Our experience also suggests that students are better off learning how to use the normal facilities of Emacs, rather than special Rutgers additions. The normal facilities are fairly good.

The Emacs commands in this document often use several "windows". For example, when you're compiling, your program will be in one window and the error messages in the other buffer. When you're debugging, your program will be in one window and the debugger in the other. Note that emacs uses the term "window" to refer to a section of your screen, and the term "buffer" to refer to a file or other object (e.g. a debugging session or a list of error messages). Normally you'll have 3 or 4 buffers active, but you'll only have 2 windows. Thus you may have to switch which buffer is in which window. The following commands will help:

*c-x o* Other window - move the cursor into the other window on the screen  
*c-x l* 1 window - remove all but one window from the screen. It doesn't get rid of any buffers. It just stops showing all but one for the moment. The buffer it leaves is the window is the one with the cursor in it.  
*c-x b* choose buffer - switch to a different buffer. It will ask you for the name of the buffer to move to, and suggest a default. To see the names of all your buffers, type "?". If you're running under X, you can also pull down the "Buffers" menu and select a buffer with the mouse.

## 2. Preparing the Source Code

You will start by using emacs (or another editor) to create a file that has your program in it. You should give it a name that shows the language it is written in. That is the name should end in .c for a C program, .p for pascal, etc. Some tools use this ending (called the "extension") to determine what language a program is written in.

language	extension	compiler
Pascal	.p	pc
C	.c	cc or gcc
Fortran	.f	f77

## 3. Compiling a Simple Program

Once you have a program, you need to compile it and build an executable. This section describes the simplest case: where your whole program is in one file. (For more complicated cases, see section section 5.)

In this section I'm going to assume that your program is in a file "prog.c". You can use any file name that ends in .c (for C) or .p (for Pascal). If you use "myname.c", just replace "prog" with "myname" in this section.

Before you compile a program, you need to create a file that defines options that are going to be used by the compiler. You could type them each time, but it's much more convenient to put them into a file once. So create a file called Makefile that contains the following:

```
CC=gcc
CFLAGS=-g -Wall -Wstrict-prototypes
PFLAGS=-g -C -H -Rw
```

**WARNING:** You must have a Makefile in each directory where you actually do your work. If you use a different subdirectory for every course or assignment, that means you'll have to put the Makefile in each one.

The CC line says to use gcc as your C compiler. On many of our systems there are two different C compilers. This just makes sure you always get the one that this document is written for.

The CFLAGS and PFLAGS lines turn on features in the C and Pascal compilers (respectively). These features allow debuggers to operate (the -g option), and check your code for errors (the -W options and -C -H -Rw). It is to your advantage to ask the compiler to do as much checking as possible. (For the experts in the audience: This setting of CFLAGS is appropriate for GCC, but not for the Sun C compiler.)

In the case of C, the two -W options cause the compiler to check for additional kinds of errors and suspicious usage. With these options, it will take you longer to get your program to compile, because the compiler will complain about more problems. But if you don't use the options, the problems will still be there -- you'll just have to find them during later debugging sessions. That will take much longer, because you won't have the compiler telling you where they are.

In the case of Pascal, the -C -H -Rw options cause the compiler to put additional runtime checking into your program. These check for things like out of bounds array references and referring to record fields that haven't been set. Again, it's much better to have the system check for these errors and tell you about them, rather than having to find them for yourself when your program mysteriously doesn't work.

To compile the program, use the emacs command "m-x compile". The first time you do this, it will ask you what command you want to use. In the minibuffer you'll see:

```
Compile command: make -k
```

You need to move the cursor into the minibuffer and add the name of your program to that command. The name of your program is the source file name without the .c or .p. So if your source file is call prog.c or prog.p, you want the command

```
Compile command: make -k prog
```

Hit carriage return once you have the command right.

You only have to type this once: Emacs will remember the command and offer it to you as the default when you type "m-x compile" in the future.

Here's what the command means:

*make* Make is a command for compiling programs. It figures out what language your program is written in and calls the appropriate compiler. It looks at settings in Makefile to help it use the right options.

*-k* This doesn't matter for a simple program, but will matter as you get into more complex assignments. If your program involves more than one source file, this tells make to continue even if there are errors in one file. That lets you find as many errors as possible at one time.

*prog* the name of the executable program you're making.

Here's what will happen

- Emacs will offer to save prog.c, if you have made changes in it that have not been saved yet.
- Emacs will call the C or Pascal compiler. This will translate your source code, which is in prog.c or prog.p into an executable binary form, and leave it in a file called prog.

At this point you could run the program using the shell command

```
./prog
```

However we're going to recommend that you run it inside emacs. See the section on debugging.

### 3.1. If There Are Errors

It's very likely that there will be an error in your program the first time you try this. If the compiler finds an error, it will issue a message. Here's what happened when I used a variable that I hadn't defined:

```
bad.c: In function `main':
bad.c:5: `ab' undeclared (first use this function)
bad.c:5: (Each undeclared identifier is reported only once
bad.c:5: for each function it appears in.)
```

Of course you may get several sets of error messages for the same program.

Emacs makes it easy to find and correct your errors. When you use the "compile" facility inside Emacs, Emacs will split the screen, putting your program in one buffer and the error messages in the other. (If for some reason this doesn't happen, use the window selection commands to get it into that state.) Put the cursor into the buffer containing your program.

You can now use the Emacs "next-error" command, and Emacs will automatically move the cursor to the place where the first error appeared. Once you've fixed it, use "next-error" again, and you'll go through the errors one by one.

You can call "next-error" either by typing "m-x next-error" or using the abbreviation "c-x '". You may find it hard to read this on the screen. That's control-X followed by a backwards single quote: '

### 3.2. More about Errors

When you compile a program, several things are happening. First, the C compiler reads your program and produces an assembly-language translation. Then the assembler reads that and produces a relocatable binary. Finally, the loader reads that, merges it with any libraries that your program needs, and produces an executable binary. Any one of these stages can produce error messages.

Error messages from the compilers usually give you line numbers in your file. Here is a typical error from the C compiler:

```
bad.c: In function `main':
bad.c:5: `ab' undeclared (first use this function)
bad.c:5: (Each undeclared identifier is reported only once
bad.c:5: for each function it appears in.)
```

Note that this shows you the name of the file and the line number within the file. Of course this is just the place where the compiler noticed the error. Often the actual mistake is on the previous line. The Emacs "next-error" command examines this information, and uses it to go directly to the appropriate place in the source file. If you're not using emacs, you would use an editor to find the specified line.

The assembler should never produce an error message. If it does, something is wrong with the compiler. An error message from the assembler will begin with "as: ".

Error messages from the loader normally indicate that you called a procedure that you

didn't supply. Here is a typical error message:

```

Undefined                               first referenced
symbol                                   in file
exp                                       /var/tmp/cca001ET1.o
ld: fatal: Symbol referencing error. No output written to test

```

This indicates that your program called a subroutine called "exp", but there is no such subroutine. This may indicate that you typed the name incorrectly, or that there was a library that you forgot to include.

For example, in C if you use a math routine (e.g. exp, sin, or cos), you have to tell the compiler that you need the math library. The math library is specified as `-lm`. To cause your programs to include the math library, add the following line to the beginning of your Makefile:

```
LDFLAGS=-lm
```

**WARNING:** The same Makefile is used for all programs in the directory. Once you add this line, all programs you build in this directory will have the math library added. That's fine if you use a separate subdirectory for each assignment, or if all your assignments use the same libraries. If you need to use different libraries for different programs, see section 5 for more information about Makefiles.

Pascal and Fortran automatically include the math library if necessary, but there are other libraries which are optional for all 3 languages (e.g. curses).

The man page for a subroutine should tell you whether a special library is needed. E.g. if you do "man exp", here's part of what you see:

```

NAME
    exp, expm1, log, loglp, logl0, pow - exponential, logarithm,
    power

SYNOPSIS
    cc [ flag ... ] file ... -lm [ library ... ]

#include <math.h>

double exp(double x);

```

This says that if you are going to use exp, you need to do two things:

- *Put*

```
#include <math.h>
```

*at the beginning of your program*

- *Use -lm when you build the program.*

Note that it's possible to need more than one library. For example, networking code often needs both `-lsocket` and `-lnsl`. To specify that, your Makefile should include the line

```
LDFLAGS=-lsocket -lnsl
```

In Fortran, a typo in an array name can also result in an error from the loader. If you type

```
x = myaray(i,j)
```

and myaray doesn't match an array declared in a dimension statement, Fortran assumes you are trying to call a subroutine. If there is no such subroutine, then you will get an error from the loader.

## 4. Running and Debugging a Program

If you simply run your program, there is a reasonable chance things will really end this way:

```
segmentation error - core dumped
```

It seems worth mentioning some of the things that can go wrong, and what to do about them.

Under Unix, most errors eventually result in some sort of memory protection error. This may show up as "segmentation fault", "bus timeout", or various other things. These problems occur because the program attempts to reference a memory location that is not part of the program. The most common cause is a pointer that has not been properly computed, or an array reference where the index value is incorrect.

Generally the only practical way to find out what is going on is to use a debugger. This document will describe Sun's debugger, dbx. The staff actually prefer another debugger, called gdb. It works fine with C. Unfortunately, 111 is currently being taught in Pascal. Gdb cannot be used to debug Pascal. Thus we're teaching you dbx. If you're using C (particularly if you're doing it at home with Linux), you'll probably prefer to use gdb. I'll supply notes where gdb differs from dbx.

**WARNING:** Dbx doesn't normally accept abbreviations for commands. When you're debugging a program, you are likely to use the command such as "step" a lot. So before you use dbx, we strongly suggest that you create a file in your home directory called .dbxrc, with the following contents:

```
set -o emacs
dbxenv suppress_startup_message 3.2
kalias n=next
kalias s=step
kalias c=continue
kalias p=print
```

This will allow you to use one-letter abbreviations for the most common commands. (Set -o emacs allows you to use emacs-style editing when you're typing commands. The dbxenv command turns out an enormous message you would otherwise get everytime you start dbx.) [Gdb has the same aliases builtin.]

## 4.1. Debugging within Emacs

We recommend that you run dbx under the control of Emacs. To do that, use the Emacs command "m-x dbx". [Use "m-x gdb" if you prefer to use gdb.] Emacs will open a second window for the debugging session, and ask you how to run dbx. It will give you a prompt starting with "dbx ". Add the name of your program. For example, suppose you've got a source file "prog.c" or "prog.p", and you produced a program called "prog". Then you'd want to make the debug command

```
dbx prog
```

If you put the debugger in one window and your source file in the other, Emacs will move around in the source file so that you can always see the code that is currently executing. In the following section, I assume that you have your source code in one window and the debugger in the other window. I also assume that your cursor is in the window with the debugger.

There are two ways to use a debugger: You can start the program and see where it breaks, or you can control it with the debugger. With the debugger, you can step through the program line by line, or place a "breakpoint" that will cause the program to stop at a particular place so you can look around.

When something goes wrong in the program, the debugger will get control. At that point you can look to see where it is, including the whole stack of subroutines that are active. You can also look at the values of variables in your program.

In many cases that's not good enough: you need to figure out what happened to lead to that state. The most common technique is to place "breakpoints" or "stop points" throughout the code. A breakpoint is an indication that when the program gets there, it should transfer control to the debugger. You can look at variables, and then continue the program. You can also step through the program a line at a time, to see where it is going.

Thus you have the following three choices when debugging:

- *You can just start the program and see what breaks. To do that, type "run" to the debugger. The "run" command simply starts your program from the beginning. If you have some idea of where the problem is, you can place a breakpoint in the program in that area. Then you can start the program with the "run" command. It will run until it gets to the breakpoint, and then stop. At that point you'll be able to look at variable values, continue a line at a time, etc. If you don't know where the problem is, you can place a breakpoint at main. Then you can start the program with the "run" command. Main is the first thing that is executed in any program. Thus this lets you step through the entire program.*

Now for some more details:

The "run" command starts your program from the beginning. If your program requires any arguments on the command line, supply those same arguments to the "run" command.

To place a breakpoint, use the "stop" command. You can break at a line or a function name. Stop at is used for line numbers, stop in for functions. For example, "stop at 4" will put a breakpoint at line 4 in the current source file. "stop in compute" will put a breakpoint at the first line in the function named "compute". [In gdb, use the command "b" for both of these. "at" and "in" aren't needed: gdb can tell the difference between a line number and a function name.] In Emacs it's usually more convenient to use function names, since line numbers can be slightly hard to find in Emacs. However if you need to place a breakpoint on a line, put Emacs in line number mode by typing "m-x line-number-mode". Then you

can put the cursor on a line in the source file, and you'll see the line number in the mode line.

When you're at a breakpoint, you can run your program a line at a time type typing "step". (This can be abbreviated as "s" if you created .dbxrc as explained above.) Emacs will put an arrow at the current line in the source file. It will move the arrow as you step through the program.

When you're stepping, sometimes you'll come to a function call. If you continue typing "step", dbx will follow the function call, and you'll end up stepping through the function. If you know that a function is OK, you can use "next" (abbreviated "n"). Then dbx won't go into the function.

Sometimes you'll realize that you're not going to see anything interesting in the current function. The "step up" command causes dbx to finish the current function without stepping. It turns stepping back on once you've exited from the function. [In gdb, use "finish" for this.]

If you want to continue your program, use the "continue" command (abbreviated "c"). Often you'll find it useful to place a breakpoint somewhere later in the program, and then continue.

If you choose to run your program and see what breaks, the most useful command is going to be "where". Once the program breaks, you'll find yourself in gdb at the point of failure. The "where" command will tell you where that is. It will also print a "backtrace" of all active function calls. You can look at the values of variables using the command "print" (abbreviated "p"). If you need to see the value of a variable in a function that called you, you can walk up the call stack using "up".

There are lots more dbx commands that I haven't described here. Dbx has a good "help" command. [So does gdb.]

To get out of dbx, use "quit". ^C won't work.

## 5. Makefiles

For a simple program, make can figure out how to compile it. But as soon as you need to build programs that have more than one source file or that need special options, you need to put additional information in your Makefile to tell make what to do.

Make is a program that runs the compiler and loader for you. Its primary purpose is to save you time. It does that in two ways:

- *It lets you put all the options you're going to use in a file, so you don't have to type them each time.*
- *It automatically figures out which files need to be compiled. If an object file is up to date, make won't bother to compile it again.*

There are three steps to using Makefiles:

- *Decide what you are going to call the program, and the various source files that go into it.*
- *Put the necessary information into a file called "Makefile".*
- *Use make to build your program.*

In this document, all of the examples use C. I'm hoping that none of your Pascal programs will become complex enough that you need this section. If you do, the only change you'd need to make in the examples is to use \$(PC) rather than \$(CC) in the command that builds the program. It's perfectly valid to use Pascal for some programs and

C for other programs in the same Makefile.

## 5.1. A Simple Example

The easiest way to explain make is to look at an example. Suppose I am doing an assignment that builds a program called "compiler". It uses three source files, "lex.c", "syntax.c" and "code.c". It requires the math library, which is called using "-lm".

I need to create a file called "Makefile" that tells make how to build the compiler. Here's an example of what should be in "Makefile":

```
CC=gcc
CFLAGS=-g -Wall -Wstrict-prototypes
PFLAGS=-g -C -H -Rw

COMPLIBS=-lm
COMPOBJS=lex.o syntax.o code.o

compiler: $(COMPOBJS)
    $(CC) -o $@ $(COMPOBJS) $(COMPLIBS)
```

Note that the first three lines form the standard Makefile that should be in every directory you use. They give you the right options for all C and Pascal programs. The rest of the file contains specifications for this one project: the compiler you're building.

As mentioned above, there is just one "Makefile" in each directory. If you're working on several different programs in the same directory, you put descriptions of all of them in one "Makefile". We'll give an example of that below.

COMPLIBS and COMPOBJS are names that you choose. I used COMP because it's a compiler project. If I were working on an editor, I might have used EDITLIBS and EDITOBS. Make sure that you use the names consistently, of course.

Now for the explanation:

```
CC=gcc
```

Specifies which C compiler to use. We recommend that you use this line just as given. That causes you to use gcc, the Gnu C compiler. To use the Sun C compiler, specify `CC=/opt/SUNWspro/bin/cc`

```
CFLAGS=-g -Wall -Wstrict-prototypes
```

Specifies what options to use for compiling C programs. We recommend that you use this line just as given. -g means to produce debugging symbols. The -W options turn on warning messages. This gets the compiler to do as much checking as possible. It makes it harder to get your code to compile correctly. But once it compiles, it's a lot more likely that your program will work.

```
PFLAGS=-g -C -H -Rw
```

Specifies what options to use for compiling Pascal programs. It's not needed for this example. It's included for consistency.

```
COMPLIBS=-lm
```

Specifies options to be used when loading the program. Normally this is simply a list of libraries required by the code. In this case, I've assumed you're going to need the math library, which is loaded using "-lm". Actually, for building a compiler, you probably don't need any libraries. In that case you'd use

```
COMPLIBS=
```

Or simply omit this line and the other mention of COMPLIBS.

[In instructions above, I indicated that libraries should be mentioned in an LDFLAGS definition. The reason I'm using a special variable COMPLIBS rather than simply including these in LDFLAGS is because I want to show you how to describe two different programs in the same Makefile. If you're only going to build one program in this directory, you could use LDFLAGS rather than COMPLIBS. In that case you wouldn't need the \$(CC) line at all, since make would be able to figure out how to build the program. You'd still need the line starting "compiler:" to tell make what files are needed for the program.]

```
COMPOBJS=lex.o syntax.o code.o
```

Specify the files that make up the compiler. Note that you specify only the .o files. Make realizes that it should look for corresponding source files ending in .c.

```
compiler: $(COMPOBJS)
          $(CC) -o $@ $(COMPOBJS) $(COMPLIBS)
```

Instructions to cause it to build the executable file "compiler". The first line lists all the object files needed for the compiler. The second line gives the actual command needed to build it. Note that you don't need to tell make how to compile the individual source files. It will automatically generate commands using CFLAGS.

WARNING: You must use a tab before \$(CC). Spaces will not work.

## 5.2. Running Make

To build "compiler", create "Makefile" as shown, and then type the command "m-x compile" inside emacs. Emacs will prompt you with a command "make -k". Add the name of the program to it, so that the final command is

```
make -k compiler
```

The -k option isn't required, but we recommend it. It simply tells make that if it finds an error in one file, it should try the other files anyway. That lets you find as many errors as possible at once.

The advantage of using Emacs is that it simplifies handling errors. Emacs will look at the error messages. You can use "next-error" to go directly to the right place in the source file.

## 5.3. A Slightly More Complex Example

As mentioned above, there's only one Makefile in each directory. If you are working on two different projects in the same directory, you put both in the same Makefile. (In fact, the example Makefile above is more complex than it needs to be. I've designed it to make it easy for you to put a second program into the same Makefile.)

The following example shows two different programs. One is almost the same as above:

it builds a program called "compiler", using source files "lex.c", "syntax.c" and "code.c". It does not need any libraries. The other program is called "gauss". It has one source file, called "gauss.c". It requires the math library, which is called using "-lm".

```
CC=gcc
CFLAGS=-g -Wall -Wstrict-prototypes

# description for "compiler"

COMPOBJS=lex.o syntax.o code.o

compiler: $(COMPOBJS)
          $(CC) -o $@ $(COMPOBJS)

# description for "gauss"

GAUSSLIBS=-lm
GAUSSOBSJS=gauss.o

gauss: $(GAUSSOBSJS)
        $(CC) -o $@ $(GAUSSOBSJS) $(GAUSSLIBS)
```

Note that comments can be added using #. As you can see, the CC and CFLAGS lines appear only once. The rest of the file contains one group for each program. You need to use different names for the LIBS and OBJS macros for each program, or there will be confusion. I have omitted COMPLIBS because the compiler doesn't need any libraries.

With this file, you'd use the command "make -k compiler" to build "compiler" and "make -k gauss" to build "gauss".