More C

Unix Programming Tools

Bill Bryant
MIT Project Athena
Revision B
September, 1987

Table of Contents

1. Introduction

The Unix operating system is a powerful programming environment. Seasoned programmers use its tools in all aspects of their work. Unfortunately, most beginning programmers have not been introduced to these tools and don't know how to use them in an integrated fashion. This document addresses this problem.

More C introduces several of the more important programming tools by tracing the development of a simple program. You will learn about:

- The C compiler and some of its options.
- The *make* utility.
- The *lint* source file checker.
- Runtime debugging and the *dbx* source code debugger.

This document also has several appendices:

- Appendix A discusses C preprocessor option to the *cc* command.
- Appendix B provides an overview of the Unix system libraries.
- Appendix C describes some useful functions from the standard C library.
- Appendix D describes some of the system's I/O functions.
- Appendix E describes functions of the system's math library.
- Appendix F explains how to create function libraries of your own.
- Appendix G provides a brief introduction to the *Saber-C* programming environment.

This document assumes that you can write simple C programs like those discussed in *Essential C*. You should know how to perform basic tasks with the Unix operating system, and you should be able to use the Emacs editor. If you don't know these basics, you must look through Athena's *Essential Unix* and *Essential Emacs*.

More C uses an example program to illustrate key points. This program, and other associated files are kept online in the */usr/doc/athena/more_c* directory. You can experiment with the program and use them to follow the tutorial examples. Use the *mkdir* command to create a subdirectory of your own, then copy the */usr/doc/athena/more_c* files into the new subdirectory as follows:

host% cp -r /usr/doc/athena/more_c/* ***~username/subdir***

where *~username* specifies your home directory, and */subdir* specifies the new subdirectory. Your subdirectory should contain the following files after the copy:

```
Makefile      Makefile3      main.c        preblank.c
Makefile2     adjstring.c    maincore.c    tblank.c
```

2. The Sample Program

This document illustrates important concepts and programming tools by tracing the development of the *blankstrip* program. This simple program takes a character string entered from the user's terminal, redisplays the input string, strips any leading and trailing blanks from it, and then displays the results. Although the program's **source code** could have been written into one file, we have broken down the program by function into several source files.

2.1 The *main.c* File

The *main*() function controls the program's execution. It prompts the user for a character string, redisplays the string, calls the *preblank*() and *tblank*() functions to modify the string, then displays the results. Here's the source code for *main.c*:

```
#include <stdio.h>
#define MAXLEN 1000 /* Maximum length for string array */

main()
{
  char instring[MAXLEN];

  printf("Enter a string with leading and trailing blanks:\n");
  gets(instring);
  printf("input = '%s'\n", instring);
  preblank(instring); /* Remove leading blanks */
  tblank(instring); /* Remove trailing blanks */
  printf("output = '%s'\n", instring); /* Print modified string */
}
```

2.2 The *preblank.c* File

The *preblank*() function removes leading blanks from the input string. Its source code is in the *preblank.c* file:

```
preblank(s)
char *s;
{
    int n = 0;

    while (s[n] == ' ' || s[n] == '\t')
    {
      ++n;
    }
    if (n > 0)
    {
      adjstring(s, n);
    }
    return;
}
```

2.3 The *adjstring.c* File

preblank() calls the *adjstring*() function to adjust the indexing of the modified string. Its source code is in the *adjstring.c* file:

```
adjstring (s, count)
char *s;
int count;
{
    int n;

    for (n = 0; s[n + count] != '\0'; ++n)
    {
        s[n] = s[n + count];
    }
    s[n] = '\0';
    return;
}
```

2.4 The *tblank.c* File

The *tblank*() function removes trailing blanks from the input string. This function uses the standard Unix function *strlen*() to determine the length of the input string. Its source code is in the *tblank.c* file.

```
tblank(s) /* This function removes trailing blanks from s */
char *s;
{
    int slength, ndex;

    slength = strlen(s); /* strlen returns length of s */

        while((s[slength] == ' ' || s[slength] == '\t'))
        {
            --slength;
        }
        s[++slength] = '\0';
        return;
}
```

3. Compiling Simple Programs

The C compiler *cc* is the utility you use to translate your C program **source code** into **machine code** that the computer can execute. Note that the filenames of all C source files must end with **.c**. If you try to compile a program with a filename that does not have a *.c* extension, the compiler complains that the file has a *"bad magic number,"* and refuses to compile the program.

To compile a simple program, use *cc* as follows:

`host%` **cc** ***source.c***

where *source.c* is the file that contains the program's source code.

If your source code has any syntactical mistakes in it, *cc* will issue error messages to that effect and will halt the compilation process. If your source code is syntactically correct, *cc* will compile it and load the resulting machine code into a file named **a.out**. *a.out* is the default name of the program's **executable file**; it is often referred to as the program's **binary file**. To run the program, type:

```
host% a.out
```

3.1 Compiling Multiple Source Files

If you are writing programs that are made up of multiple source files, use *cc* as follows:

`host%` **cc** ***source1.c source2.c source3.c*** **`. . .`**

cc will compile the source files and load them all into the *a.out* file.

To compile the example *blankstrip* program, type:

`host%` **cc main.c preblank.c adjstring.c tblank.c**

cc displays the following as it compiles the program:

```
main.c:
preblank.c:
adjstring.c:
tblank.c:
```

When *cc* finishes, run the program by typing *a.out*. The program will prompt you with the following message:

```
Enter a string with leading and trailing blanks:
```

Do as it says, but don't be suprised if the program doesn't work as advertised. We'll get around to fixing it later.

4. The Compilation Process

The C compiler is a versatile programming tool, more versatile than you might realize. But you won't be able to do much with *cc* until you understand how the command goes about compiling a program.

Actually, the command name *cc* is a misnomer. The C compiler does much more than "compile" programs. It is more accurate to say that *cc* "constructs" them.

4.1 The C Compiler's Three Phases

The C compiler constructs a program in three phases:

1. *cc* **preprocesses** the source code with the **Macro Preprocessor**.
2. *cc* compiles the source code into **object code**.
3. *cc* uses the **Link Editor** to **load** the object code into the *a.out* file, and to **link** the different parts of that file together so that *a.out* can be executed.

When *cc* constructs a simple program, one that has all its source contained in one file, the utility preprocesses, compiles, and loads in one go. If, on the other hand, the program has more than one source file, *cc* preprocesses and compiles each source file separately, creating for each source file a corresponding **object file**. When it finishes processing the source files, *cc* then passes on to the loading and linking phase. See *figure 4-1* below.

4.2 The Macro Preprocessor

Before it compiles a source file into an object file, *cc* uses the macro preprocessor to check for and process any embedded preprocessor commands. (Remember, as explained in *Essential C* that all preprocessor commands begin with the sharp or pound sign **#**; this sign must be placed in the first column of the source file.)

The sample program has two preprocessor statements, both of them contained in the *main.c* file:

```
#include <stdio.h>
#define MAXLEN 1000
```

These are typical preprocessor commands. The first command tells the preprocessor to insert the system header file *stdio.h* into the source file at the point of the *include* statement. Note that this header file itself contains preprocessor commands, which will in turn be processed.

The second command in *main.c* is a substitution statement. The preprocessor will replace any occurrence of MAXLEN within *main.c* with the value 1000, as long as the MAXLEN is not part of a quoted string. For instance, this MAXLEN gets replaced:

Figure 4-1: The Compilation Process

```
char instring[MAXLEN];   to   char instring[1000];
```

This one remains unchanged:

```
printf("My favorite symbolic constant is MAXLEN\n");
```

4.2.1 Header Files and the *#include* Statement

In general, the preprocessor looks in two places for header files: the system's standard header directory */usr/include*, and the source file's directory. If the specified file is enclosed in double quotes:

```
#include "header.h"
```

the preprocessor searches the *source file's directory* first. If it doesn't find the file there, it then searches */usr/include*.

In cases where the specified file is enclosed in pointy brackets:

```
#include <header.h>
```

the preprocessor searches */usr/include* without bothering to search the current directory.

Note that you must enclose header filenames with either the the double quotes or the pointy brackets. If you don't, the macro preprocessor will display something like the following:

```
main.c: 2: bad include syntax
*** Exit 1
```

4.2.2 Subdirectories within *<u></u>/usr/include*

Some of the system's standard header files are in subdirectories within the */usr/include* directory. The *X window system* header files are a notable example. If you use the *ls* command on the */usr/include/X* directory, you get a list of the following files:

```
X.h             Xproto.h          rgb.h
XMenu.h         Xtext.h
Xkeyboard.h     Xtty.h
Xlib.h          mit-copyright.h
```

If you need to include the *Xlib.h* file into a program, use the *include* statement like this:

```
#include  <X/Xlib.h>
```

4.2.3 Preprocessor Options to the C Compiler

The *cc* command has several options that let you manipulate the preprocessor. If you develop programs that have their own header files stored in subdirectories, you can use the **-I** option to include them at compile time. And you can use the **-E** option to watch what the preprocessor does as it works on your source files. See Appendix A for more information about these options.

4.3 Object Files

In the second phase of construction *cc* compiles the preprocessed source file into object code and deposits the results in an object file. All object files end with a **.o** extension. For instance, when *cc* compiles the *main.c* file, it creates a *main.o* object file.

4.3.1 Functions and the Object File's Symbol Table

C program source files consist of function calls and function definitions. For instance, take another look at the *main.c* file:

```
#include <stdio.h>
#define MAXLEN 1000 /* Maximum length for string array */

main()
{
  char instring[MAXLEN];

  printf("Enter a string with leading and trailing blanks:\n");
  gets(instring);
  printf("input = '%s'\n", instring);
  preblank(instring); /* Remove leading blanks */
  tblank(instring); /* Remove trailing blanks */
  printf("output = '%s'\n", instring); /* Print modified string */
}
```

This file defines the program's *main* function, which in turn calls four other functions: *printf*(), *gets*(), *preblank*(), and *tblank*(). Note that none of the four functions are defined in the *main.c* file. These function calls are **external references**; they call functions that are defined outside of the *main.c* file.

When *cc* translates a source file into an object file, the compiler creates, as part of the file, a **symbol table**. This table contains two lists: a list of functions defined, and a list of external references. *main.o*'s symbol table has the following entries:

Functions Defined	External References
main()	printf()
	gets()
	preblank()
	tblank()

These entries are used by the link editor *ld* when that utility loads the *main.o* file into *a.out*.

4.4 The Link Editor *ld*

After *cc* has compiled every source file into its corresponding object file, it calls the link editor *ld* to load the object files, search the C library and load the appropriate library functions, then make the program executable. *ld* is sometimes referred to as the **loader**.

ld is not part of the compiler; *cc* merely calls the link editor when a program's compiled pieces are ready to be put together into one executable file. Because *cc* calls *ld* automatically, **you do not need to use the *ld* command yourself.** Experienced programmers will, on occasion, use *ld* separately, but those situations are beyond the scope of this document.

ld performs four tasks when it assembles a program:

- It loads the program's object files into the *a.out* file.
- It searches the C Library for missing functions and loads the appropriate ones.
- It links the loaded object files and functions together.
- If no errors have occurred, and the *a.out* file is complete, *ld* makes *a.out* executable.

4.4.1 Loading the Object Files

The link editor loads the object files into *a.out* by concatenating them in the order their associated source files appeared in the *cc* command line. Suppose you compile the sample program like this:

```
host% cc main.c preblank.c adjstring.c tblank.c
```

The link editor will load the object files in this order:

```
main.o  preblank.o  adjstring.o  tblank.o
```

4.4.2 Format of the *a.out* File

As shown in *figure 4-2*, the *a.out* file has five sections:

1. The **header** section, which lists the sizes of the other sections.
2. The **text and data** section, which contains in binary form the program's instructions and data.
3. The **relocation information** section.
4. The **symbol table** section.
5. The **strings table** section.

Figure 4-2: Format of the *a.out* File

For the purposes of this discussion, you can assume that the last three sections, the relocation information, the symbol table, and the strings table, are all one section: the symbol table section. If you want to know the nitty-gritty details, read the manual pages for *ld* and *a.out*.

4.4.3 Loading the Objects and Creating Symbol Table Entries

When *ld* loads an object file into *a.out*, the object code is placed in the *text* section of *a.out*, while the object file's symbol table information is placed in the *a.out*'s symbol table. As *ld* loads the object code, it writes the following information into its symbol table:

- The location within *a.out* of all function definitions.
- The location within *a.out* of all function calls.

The link editor will use this information after the loading process to link the function calls to the function definitions.

Figures 4-3 and *4-4* trace the loading of the object files and the creation of the symbol table entries. First *main.o* is loaded, then *preblank.o*, then *adjstring.o*, then *tblank.o*:

Figure 4-3: Loading *main.o* and *preblank.o*

Figure 4-4: Loading *adjstring.o* and *preblank.o*

4.4.4 Loading Functions from the C Library

Now *ld* searches the C library for the symbol table's remaining *external references*, those functions that are called within *a.out*, but have yet to be loaded. If you examine the second symbol table in *figure 4-4*, you will find that three functions remain unloaded: *printf*(), *gets*(), and *strlen*(). *ld* finds all three of these functions in the C library, loads them into *a.out*, and updates the symbol table accordingly, as shown in *figure 4-5*.

By locating and loading the library functions, *ld* has resolved all the external references listed in the symbol table. Each function call can now be matched to a loaded function definition. *a.out*'s text segment and symbol table are now complete. See *figure 4-6*

If, at this point, the symbol table still contained function calls that could not be matched to function definitions, *ld* would have displayed an error message that looks something like this:

```
Undefined:
_func_name1
_func_name2
        .
        .
```

where the *func_name*s specify the names of the missing functions.

4.4.5 The Link Editor Edits *a.out*

Now that the symbol table is complete, *ld* can determine the location of every function definition and every function call. *ld* links the calls to the definitions by replacing each function call with information that specifies the location of the function definition. For instance, the *printf*() function call that is made at address 3 is replaced by a pointer to address 30, the location at which *printf*() is defined. *Figure 4-7* shows *a.out* completely linked.

4.4.6 Making *a.out* Executable

As its last step, *ld* changes *a.out*'s **file protection mode** so that the system will treat the file as an executable when you enter the filename. Use the *-l* option to *ls* to look at *a.out*'s file protection mode:

```
host% ls -l a.out
-rwx--x--x  1 usename        8777 Jan 24 05:58  a.out
```

The first field in the listing specifies the protection mode. The *x*'s tell the system that *a.out* is an executable file. When you enter the filename *a.out*, the system runs the program.

If *ld* is unable to resolve all of the program's external references, it will not make *a.out* executable. Instead, the file's protection mode is set to this:

```
-rw------   1  usename       8777 Jan 24 05:58  a.out
```

No *x*'s here. When you try to execute an *a.out* such as this, the system refuses to cooperate:

Figure 4-5: Loading the C Library Functions

Figure 4-6: *a.out* Completely Loaded

Figure 4-7: Linking *a.out*

```
host% a.out
a.out: Permission denied.
```

4.5 Options to the C Compiler

One glance at the *cc* manual page reveals an astonishing number of options. Now that you (hopefully) understand what the compiler actually does when you run it, the manual page might be a little easier to understand. The rest of this chapter introduces to two of *cc*'s most commonly used options: *-c* and *-o*.

Other *cc* options are discussed in the appendices of this document. Appendix A describes *cc* options that let you manipulate the macro preprocessor. Appendix B provides a summary of the C libraries, and shows you how to access other libraries with the *-l* option.

4.5.1 Using *cc* without the Link Editor

There are times when you want to compile source files without bothering to create *a.out*. You may want to compile pieces of a program to make sure they are syntactically correct , but you don't want to, or aren't able to compile the entire program. The **-c** option to *cc* tells the compiler to compile the source files into object files, and to stop right there.

Use the option as follows:

```
host% cc -c source.c
```

This command produces a *source.o* file.

You can also use the option to compile multiple source files into multiple object files:

```
host% cc -c source1.c source2.c ...
```

This command produces the files *source1.o*, *source2.o*, and so on.

4.5.2 Using *cc* to Load Precompiled Object Files

The C compiler can distinguish between program source files and object files that have been compiled with the *-c* option. (It recognizes the difference between *.c* and *.o* files.) If you ask *cc* to process object files, the utility will skip the preprocessor and compilation phases, and proceed directly to the loading and linking phase. So you could, if you wanted to, compile the *blankstrip* program with two commands, one to preprocess and compile the source files, the other to load the objects into *a.out*:

```
host% cc -c main.c preblank.c adjstring.c tblank.c

main.c:            |
preblank.c:        |   Preprocess and compile the source files.
adjstring.c:       |
tblank.c:        __|

host% cc main.o preblank.o adjstring.o tblank.o        Link and load the objects.
```

You can even compile source files and object files at the same time. Suppose, for instance, that you have just corrected a problem with the source file *preblank.c*. You can reconstruct *blankstrip* by compiling the updated source file and the object files at the same time:

```
host% cc main.o preblank.c adjstring.o tblank.o
```

The command preprocesses and compiles *preblank.c* first, then loads the object files into *a.out*.

4.5.3 Renaming *a.out* with the *-o* Option

Each time you compile and load a program with the *cc* command, *ld* creates an *a.out* file. If your current directory already has an *a.out* file, the link editor will overwrite it with the new *a.out*. You can use the *-o* option to *cc* to change the default *a.out* filename to a name of your choosing.

Use the *-o* option by typing:

```
host% cc source.c -o executefile
```

where *executefile* is the name you choose for the executable version of the program. The *-o* option can be used when you compile programs made up of several source files. To compile and load the sample program into an executable file called *test*, use *cc* like this:

```
host% cc main.c preblank.c adjstring.c tblank.c -o test
```

4.6 Testing the Program

If you haven't done so already, go ahead and test the program:

```
host% test
Enter a string with leading and trailing blanks:
```

In response to the prompt, enter a string with four leading and four trailing blanks, something like the following:

```
    asdf    
```

The program responds as follows:

```
input = '    asdf    '
output = 'asdf    '
host%
```

The program doesn't work properly. It strips away the leading blanks all right, but fails to remove the trailing ones. We'll have more to say about this in the coming chapters.

5. The *make* Utility

The C Compiler is an important part of the C programming environment in Unix, but in the course of developing *real* C programs, seasoned programmers do not use the *cc* command to create their executables. Instead, they **make** their programs with the **make** utility.

The *make* utility is not a compiler. It is a compiler manager. When you use *make* to compile a program, the utility performs the compilation with *cc*. *make* can be used to compile Fortran and Pascal programs as well, and you can even use it to format large documents with the *scribe* text formatter.

When you develop a *real* C program, you can't expect to write the source code, compile it, and have it run perfectly the first time. The development process usually includes a lengthy debugging cycle:

1. You test the program and think about its problems.
2. You edit the source code and recompile.
3. You test the program again.

This can become a tedious cycle, especially if you have to keep entering cumbersome command lines like the following:

```
host% cc main.c func1.c func2.c func3.c -o program
```

If you have set up your program so that it can be compiled using *make*, you can rebuild the program by simply typing:

```
host% make
```

There are, however, more compelling reasons for using *make*, reasons that you may not have the background to appreciate at the moment. So, without further ado . . .

5.1 The Background Information

The nature of the debugging process is such that you can often trace bugs quickly to specific source files. Consider for example the sample program. If you will recall, when you tested the program at the end of the preceding chapter, it stripped the leading blanks properly, but failed to remove the trailing ones:

```
host% test
Enter a string with leading and trailing blanks:
   asdf
input = '    asdf     '
output = 'asdf     '
host%
```

The program uses two functions to remove blanks: *preblank*() removes the leading blanks, *tblank*() removes the trailing ones. Since the trailing blanks are not being stripped, it seems reasonable to assume that the *tblank*() function has a bug in it.

We will examine the nature of debugging in the next chapter, and in the course of that discussion, you will isolate and correct the bug in *tblank*(). However, for the purposes of the current discussion, assume that you have already debugged the function, have edited the faulty code in *tblank.c*, and have written the file to disk. Suppose that you are now ready to recreate the program's binary and test your modifications.

5.1.1 Rebuilding Binaries Efficiently

To rebuild the program, you could compile every source file just as you did the first time you compiled the it:

```
host% cc main.c preblank.c adjstring.c tblank.c -o binary
```

This is not, however, the most efficient way to rebuild the program. Note that when you initially created *binary*, the compilation process created object files for each of the program's source files. Unless you have explicitly removed these files with the *rm* command, they still exist within your development directory:

```
host% ls *.o
adjstring.o     main.o          preblank.o      tblank.o
```

In this round of debugging, *tblank.c* is the only source file you modified; you did not change *main.c*, *preblank.c*, or *adjstring.c*. Since *tblank.c* was the only file modified, it is the only file that must be recompiled. The other object files are **up-to-date** with respect to their associated source files, and they can be loaded into the executable file as is. Hence, you can rebuild the program as follows:

```
host% cc -c tblank.c
host% cc main.o preblank.o adjstring.o tblank.o -o binary
```

Or alternatively:

```
host% cc main.o preblank.o adjstring.o tblank.c -o binary
```

In the case of this program, the time you save loading up-to-date objects seems negligible when compared to the time it takes to recompile the entire program. But that's only because the sample program is such a simple one. The time you save when rebuilding a *real* program can be considerable.

Consider what happens when you compile the Unix *sail* program from its sources. This popular wargame program has 23 source files. Using an Athena workstation, I compiled the entire program and recorded the compilation time with the *time* command. The operation consumed 51.5 seconds of CPU time (the time spent by the workstation's central processor), and I waited more than seven minutes for the compilation to complete. I then rebuilt the program by recompiling only one of the source files, then loading the program's objects. This operation consumed 4.8 seconds of CPU time, and I only had to wait 39 seconds for the rebuild to complete. Quite a difference.

5.1.2 Keeping Track of What Needs to be Recompiled

Of course you can't rebuild a program efficiently, or correctly for that matter, if you don't keep track of what files need to be recompiled. This isn't a serious problem in the case of our toy *blankstrip* program, but there are cases where the problem is a significant one.

Suppose you were working on a program that had several source files that depended on a certain header file. Suppose that you changed information within that header file, and now want to rebuild the program. Each source file that includes the header file must be recompiled; if you forget to compile one of them, you won't have updated the program properly.

5.1.3 *make*'s Raison d'Etre

The *make* utility automates the efficient rebuilding of programs from source. It keeps track of what files depend on one another, and when asked to rebuild a program, it compiles only those files that have to be compiled. In the realm of *real* Unix programming, *make* is as indispensable as the compilers it manages.

5.2 *Makefile*, not Magic

The *make* utility is indispensable, but it isn't magical. It cannot rebuild programs out of thin air. The utility must have a set of instructions from which to work. The file that holds these instructions is known as the **Makefile**.

In its simplest form, a makefile tells *make* three things:

- **What** to build. This *what* is referred to as a makefile **target**.
- What the **pieces** are required for the construction of the *target*. These are the source files.
- How to build the *target*. What compilation command to use.

As you may have noticed when you copied the tutorial directory into your own workspace, you copied in several sample makefiles: *Makefile*, *Makefile2*, and *Makefile3*. By convention, makefiles are named either *makefile* or *Makefile*; many programmers perfer *Makefile* so that when they list their development directories, the uppercase *M* filename gets listed before the lowercase source file names.

5.2.1 Makefile Format

Go ahead and look at the sample *Makefile*; you will find it has the following contents:

```
# This Makefile tells how to construct the blankstrip program.

blankstrip: main.o  preblank.o  adjstring.o  tblank.o
        cc main.o preblank.o adjstring.o tblank.o -o blankstrip
```

(The line numbers are not part of the file.)

The first line in the file is a comment line. *make* ignores any text that follows a # character. Also note that the utility ignores blank lines.

The rest of the file tells *make* how to build the *blankstrip* program. Line three is a **target dependency** instruction. *blankstrip* is the *target*, the program's executable file. Becauses the target cannot be built until the program's object files have been compiled, you can say that *blankstrip* **depends** on the existence of *main.o*, *preblank.o*, *adjstring.o*, and *tblank.o*.

The target dependency instruction has an associated *command sequence* instruction, the compilation command on line four. The command sequence instruction tells *make* what commands to execute to build the target. Note that the command sequence instruction has been indented with a `Tab` character. All command sequences must begin with a `Tab` character because that is how *make* distinguishes between target dependency instructions and their associated command sequences.

5.2.2 Why Base Dependency on the Object Files?

Some of you may have wondered why we specified object files in the target dependency instruction. Certainly *blankstrip* depends upon the existence of the source files as much as it depends on the object files. Why not list the dependency files as *main.c*, *preblank.c*, *adjstring.c*, and *tblank.c*? Because *make* is capable of drawing certain conclusions based on the information it finds in the makefile. The utility knows that object files cannot be created without the presence of source files. It assumes that if a *main.o* file has been specified in the makefile, a corresponding *main.c* file exists in the current directory. And the utility knows how to compile source files into object files without having to be told the explicit commands.

To illustrate what I mean, remove the object files in the program directory:

```
host% rm *.o
```

Now that the objects have been removed, *make* has to build the program from its sources. Go ahead and *make* the program:

```
host% make
cc  -c  main.c
cc  -c  preblank.c
cc  -c  adjstring.c
cc  -c  tblank.c
cc  main.o  preblank.o  adjstring.o  tblank.o  -o  blankstrip
host%
```

Here's how *make* built the program:

1. The utility read the instructions in *Makefile* that told it how to build *blankstrip*. The instructions also specified that the target depended upon certain object files.

2. *make* searched the current directory for the specified object files and discovered that they did not exist. The utility then searched the directory for

corresponding source files and found them.

3. The utility then compiled each source file into the appropriate object file with the **cc -c** command, displaying each command line as executed.
4. Having built the missing objects, *make* then used the command sequence instruction specified in *Makefile* to finish building the program, again displaying that command line as executed.

5.3 *Make* and Modification Dates

Now that you have built the program from sources, try remaking the program:

```
host% make
'blankstrip' is up to date.
```

make won't rebuild the program because it is up to date; you haven't changed it since the last rebuild.

Make uses **file modification dates** to determine whether or not a program is up to date. Every time you create or edit a file, and write that file to disk, the Unix filesystem updates the file's *modification date* to reflect the time at which it was written.

You can examine file modification dates by using the *-l* option to the *ls* command. The following command displays the *long format* listing for the program's source and object files, and its executable file *blankstrip*:

```
host% ls -l blankstrip *.c *.o
-rw-------  1 username         174 Jan 14 12:29 adjstring.c
-rw-------  1 username         147 Jan 14 12:36 adjstring.o

-rwx--x--x  1 username       10240 Jan 14 12:36 blankstrip

-rw-------  1 username         4o7 Jan 14 12:29 main.c
-rw-------  1 username         426 Jan 14 12:36 main.o

-rw-------  1 username         167 Jan 14 12:29 preblank.c
-rw-------  1 username         173 Jan 14 12:36 preblank.o

-rw-------  1 username         281 Jan 14 12:29 tblank.c
-rw-------  1 username         172 Jan 14 12:36 tblank.o
```

(I have added the blank lines to make the example legible.)

The date and time listed next to each filename specifies the file's modification date (your dates will differ of course). Note that all the source files have older modification dates than their associated objects. In my listing *adjstring.c* is seven minutes older that *adjstring.o*, *main.c* is older than *main.o*, and so on.

An object file that is younger than its source file is considered *up-to-date* with respect to that source file because it has been compiled subsequent to the source file's last update. Conversely, an object file that is older than its source file is not *up-to-date*; the source file has been changed since the last time it was compiled.

When you ask *make* to rebuild a program, the utility decides what pieces of the program need recompiling by comparing the modification dates of the sources with the objects. If all of the object files are up-to-date, *make* tells you that the program is up-to-date, and then halts. If certain object files are out of date, *make* recompiles the corresponding source files, then rebuilds the program.

5.3.1 Playing with *make*

You can test *make* by simulating the modification of one of its source files, say *tblank.c*. Do so by entering the following command:

```
host% touch tblank.c
```

The *touch* command updates a file's modification date without changing its contents.

By *touching* the source file, you have made *tblank.c* younger than its object counterpart. If you want to confirm the change, use the *ls -l* command and examine the new modification date. Try remaking the program again:

```
host% make
cc  -c  tblank.c
cc  main.o  preblank.o  adjstring.o  tblank.o  -o  blankstrip
```

The utility recompiles the "modified" source file, then rebuilds the program from the objects.

5.4 Using *make* Macro Definitions

Most experienced programmers make liberal use of the *make* utility's **macro definition** facility. *Macro definitions* are used to create abbreviations for, among other things, long lists of filenames.

Take another look at *Makefile*:

```
# This makefile tells make how to construct blankstrip

blankstrip:  main.o  preblank.o  adjstring.o  tblank.o
     cc  main.o preblank.o adjstring.o tblank.o  -o  blankstrip
```

Both the target dependency instruction *and* the command sequence contain a list of the program's object files. Using *make*'s macro facility, you can clean up the file so that it looks like this:

```
# This makefile tells make how to construct blankstripper

OBJECTS = main.o  preblank.o  adjstring.o  tblank.o

blankstrip: $(OBJECTS)
        cc  $(OBJECTS)  -o  blankstrip
```

(These are the contents of the file *Makefile2*)

OBJECTS is the name of the macro, and is set to the value:

```
main.o  preblank.o  adjstring.o  tblank.o
```

To use the macro, you must prefix it with the **$** character. Since the macro's name is longer than a single character, you must also place it within parentheses. You can use curly braces ({ }) as well. Now when *make* reads *Makefile*, it replaces all instances of

```
$(OBJECTS)
```

with

```
main.o  preblank.o  adjstring.o  tblank.o
```

To test the *make* macro, use the following commands to replace *Makefile* with *Makefile2*:

```
host% mv Makefile Makefile.old
host% mv Makefile2 Makefile
```

Next remove all the object files so that *make* has to rebuild the program from sources. Then *remake* the program:

```
host% rm *.o
host% make
cc  -c main.c
cc  -c preblank.c
cc  -c adjstring.c
cc  -c tblank.c
cc main.o preblank.o adjstring.o tblank.o -o blankstrip
```

In addition to helping you tidy up makefiles, macro definitions provide a convenient way to change things within makefile with a minimum of fuss. Just add or remove the name to or from the appropriate macro definition, and *make* will substitute properly throughout the rest of the file.

5.5 Other Targets, Other Applications

So far, the example *Makefile*s have done nothing more than build and rebuild the program. The *make* utility can help you manage other aspects of a program's development. You can use *make* to print listings of your source files, to clean the development directory of extraneous files, and to install completed software in the right locations.

Consider the following makefile, *Makefile3*:

```
# This makefile tells make how to construct and install blankstrip

OBJECTS = main.o preblank.o adjstring.o tblank.o
SOURCES = main.c preblank.c adjstring.c tblank.c
INSTALLDIR = $(HOMEDIR)/bin

blankstrip: $(OBJECTS)
	cc $(OBJECTS) -o blankstrip

clean:
	rm $(OBJECTS)

print:
	lpr $(SOURCES) Makefile

install: blankstrip
	install blankstrip $(INSTALLDIR)

debug:
	cc -g $(SOURCES)

lint:
	lint $(SOURCES)
```

Our previous makefiles had one target, the executable file *blankstrip*; this *Makefile* has five additional targets: *clean*, *print*, *install*, *debug*, and *lint*. These targets are not filenames, they are what I call task names. Instead of building these targets, *make* performs these tasks. Note that, with the exception of the *install* target, the new targets do not depend on any of the program's files. The rest of this chapter discusses the *clean*, *print*, and *install* targets. The *lint* and *debug* targets will be discussed later.

How does *make* distinguish between targets when you run the utility? The rule works like this: if you execute *make* without specifying a target, the utility assumes that you want to create the first target listed in the *Makefile*, and proceeds accordingly. If you want to *make* any of the other targets, you must provide their names on the command line.

As you did earlier, move files around so that *Makefile3* becomes the new *Makefile*. That done, you can now experiment with the new *Makefile*. First clean the directory:

```
host% make clean
rm main.o preblank.o adjstring.o tblank.o
```

To make sure that the object files were removed, try remaking *blankstrip*:

```
host% make
cc  -c main.c
cc  -c preblank.c
cc  -c adjstring.c
cc  -c tblank.c
cc main.o preblank.o adjstring.o tblank.o -o blankstrip
```

They were indeed removed.

You can ask *make* to work on more than one target at a time:

```
host% make clean print
rm main.o preblank.o adjstring.o tblank.o
lpr main.c preblank.c adjstring.c tblank.c Makefile
```

5.6 The *install* Target

If you are writing a *real* program, one that you would like to distribute to people who use other systems and machines, you will want to send a *Makefile* that builds the software properly, and installs it in the appropriate directory. Hence the *install* target.

Here are the target's dependency and command sequence lines, preceded by the line that defines the *INSTALLDIR* macro:

```
INSTALLDIR = $(HOMEDIR)/bin

install: blankstrip
        install  blankstrip  $(INSTALLDIR)
```

The *install* target depends upon the existence of the *blankstrip* file; the software cannot be installed unless it has previously been built. The command sequence uses the Unix *install* utility to install the software properly.

In its simplest form, The *install* utility copies the specified binary file into the specified *destination* directory, and by default, makes the binary world-executable. For more information about the *install* utility, type *man install*.

Our *Makefile*'s *install* command copies the *blankstrip* binary into the directory specified by the *INSTALLDIR* macro. Note that the macro contains as part of its definition another macro: *HOMEDIR*. The *HOMEDIR* macro is not defined in the *Makefile*; it is an **externally** defined macro, one that you define when you enter the *make* command. This macro lets your friends install the software within their home directories without having to edit the *Makefile*.

To define macros externally, use the *make* command with the following format:

```
host% make  MACRO=macro_def  target . . .
```

For example, I would use the following command to install *blankstrip* into my home directory on my workstation:

```
host% make HOMEDIR=/mit/username install
install  blankstrip  /mit/username/bin
```

Do not put spaces between the macro name, the equal sign, and the macro definition, because *make* will misunderstand:

```
host% make HOMEDIR = /mit/username install
Make:  Don't know how to make HOMEDIR.  Stop.
```

5.6.1 *install* does the Right Thing

When I installed *blankstrip* in the previous example, the binary had already been rebuilt. Will *make* install the software properly if I simulate the editing of one of the source files?

```
host% touch main.c
host% make HOMEDIR=/mit/username install
cc -c main.c
cc main.o preblank.o adjstring.o tblank.o -o blankstrip
install blankstrip /mit/username/bin
```

Yes indeed.

Since the *install* target depends upon *blankstrip*, *make* first checks to see if the binary file needs updating. The utility matches modification times and notes that *main.c* has a newer date than its corresponding object file. *Make* recompiles, *main.c*, reloads *blankstrip*, then installs the software as specified by the *install* command sequence.

5.7 Options to *make*

The *make* command has several options, three of which I'll describe briefly.

5.7.1 The *-n* Option

This option tells *make* to trace and print the commands needed to update the target, but to do so without executing the commands. The following sequence of commands will illustrate:

```
host% touch main.c tblank.c          <-- Update the source files.

host% make -n                        <-- What will make do?
cc -c main.c
cc -c tblank.c
cc main.o preblank.o adjstring.o tblank.o -o blankstrip

host% make                           <-- Make the program.
cc -c main.c
cc -c tblank.c
cc main.o preblank.o adjstring.o tblank.o -o blankstrip
```

5.7.2 The *-f* Option

This option lets you specify an alternate name for the makefile:

```
host% make -f filename
```

For instance, you could have used this option to get to the various makefiles used in this chapter:

```
host% make -f Makefile3
```

5.7.3 The *-k* Option

This option tells *make* to continue processing a target, even if a step within that process fails. What does that mean? An example. Suppose you have edited *blankstrip*'s *main.c* and *tblank.c* files. Since you have updated two files, *make* should recompile them both before reloading the program. But suppose you have inadvertently left out a semi-colon somewhere in *main.c*. This is what happens when you attempt to *make* the program:

```
host% make
cc  -c main.c
"main.c", line 11: syntax error
"main.c", line 11: illegal character: 134 (octal)
"main.c", line 11: cannot recover from earlier errors: goodbye!
*** Exit 1

Stop.
```

After discovering the error in *main.c*, *make* does not attempt to compile *tblank.c*.

But this is what happens when you use the *-k* option:

```
host% make -k
cc  -c main.c
"main.c", line 11: syntax error
"main.c", line 11: illegal character: 134 (octal)
"main.c", line 11: cannot recover from earlier errors: goodbye!
*** Exit 1
cc  -c tblank.c                         <-- tblank.c gets compiled.
'blankstrip' not remade because of errors
```

make continues processing *tblank.c* even though the compiler has choked on *main.c*. (Of course *blankstrip* does not get built because *cc* could not produce *main.o*.)

5.8 Using *make* from within the Emacs Editor

You can edit source files, then compile or *make* them without having to leave the editor. Emacs even has a utility for parsing compiler error messages and moving the cursor to source code lines that generated the problems. For more information, see Chapter 22 of the *GNU Emacs Manual*. You can find this manual in the documentation racks of most Athena clusters. Also look at the Athena document *More Emacs*.

5.9 WARNING: File Transfer and *make*

Because the *make* utility depends on file modification dates to build things properly, you have to be careful when transferring makefiles and source files from machine to machine. Sometimes a transfer changes modification times inconsistantly and *make* is unable to adjust.

If you use *rcp* to transfer files, consider using that command's *-p* option. This option attempts to perserve a file's modification date so that the date it had on the sending machine matches the date it has on the receiving one.

5.10 Other *make* Documentation

Of course *make* has its own manual page in the Unix documentation. Just type **man 1 make**. You should also read the article Make - A Program for Maintaining Programs. This document can be found in Volume 1 of the *UNIX Programmer's Supplemental Documents Manual*, which in turn can be found in the documentation racks at most Athena clusters.

6. *lint*: the C Program Checker

When *cc* compiles a source file, it checks the source code for errors in syntax. A syntax error occurs when source code violates the rules of the C language. If a source file has syntax errors, it cannot be compiled successfully.

Unfortunately, a program can pass *cc*'s syntactical tests and still have what might be considered syntactical problems. For instance, the compiler doesn't care if you declare variables that do not get used. It doesn't care if you call functions with inappropriate arguments. You can supply a variable declared as *long* to a function expecting a *char*, and *cc* will not complain.

Because *cc* isn't as careful as it could be, many programmers use the **lint** utility to check their source files. *Lint* is a C source code checker that can help you locate potential troublespots in your work. It looks for inefficient code, mismatched arguments to functions, and **unportable** C expressions, expressions that may not work properly when the program is compiled on other machines.

lint **only checks source files, it does not compile them.** You may not want to use *lint* every time you compile a program. When you're writing short, simple programs, *cc* will catch the important problems. There are three cases when *lint* is very useful:

1. *cc* produces an unintelligible error message when you try to compile. *lint*'s error messages are sometimes clearer than *cc*'s. Use *lint* on source files with troublesome compiletime errors.

2. Your program produces a strange runtime bug. *lint* can sometimes locate potentially "buggy" features in a program. *lint* cannot, however, check the logic of a program's algorithm.

3. You are going to distribute software to other people who might run it on different kinds of hardware. If a program passes through *lint* without producing any errors, the chances are good that it can be compiled and executed on another person's machine.

6.1 Using *lint* on the *blankstrip* Program

In the previous chapter, the final *Makefile* contained the following lines:

```
lint:
        lint $(SOURCES)
```

with *SOURCES* defined as follows:

```
SOURCES = main.c preblank.c adjstring.c tblank.c
```

Lint is a task target that you can execute with *make*:

```
host% make lint
```

Lint displays the following when you enter the *make* command:

```
lint  main.c  preblank.c  adjstring.c  tblank.c
main.c:
preblank.c:
adjstring.c:
tblank.c:
tblank.c(4): warning: ndex unused in function tblank
gets returns value which is always ignored
```

The first error message indicates that the variable *ndex* has been declared in the *tblank*() function, but not used. According to the message, this declaration is made on line four of the source file (tblank.c(**4**)). When the computer runs the *blankstrip* program, it allocates space in memory for the *ndex* variable, but that space does not get used. You can make the program more efficient by removing the variable's declaration. Change line four in *tblank.c* from this:

```
int  slength,  ndex;
```

to this:

```
int  slength;
```

The other *lint* message refers to a problem with *main*()'s call to the *gets*() function. *Main()* ignores the value returned by *gets*(). If you look at the manual page for *gets*() (type **man 3 gets**), you find that the function is supposed to return a character string variable to the calling program. By ignoring the value returned by *gets*(), *main*() does not check that the function returns a bona fide string.

This points out a serious flaw in the program: *main*() will continue execution even if the call to *gets*() fails to produce a character string. For instance, when you run the program and enter a **Ctrl-d** in response to the prompt (hold down the **Ctrl** key and type a **d**), *blankstrip* proceeds as if you had entered a character string:

```
host% blankstrip
Enter a string with leading and trailing blanks
Ctrl-d
input = ''
output = ''
```

The *Ctrl-d* sequence produces an **end-of-file (EOF)** signal, not a character string.

To fix the program, edit the source code in *main*() so that it checks the return value of *gets*() before continuing about its business:

```
#include <stdio.h>
#define MAXLEN 1000 /* Maximum length for string array */

main()
{
  char instring[MAXLEN];

  printf("Enter a string with leading and trailing blanks:\n");

  if( gets(instring))            <--- If gets() returns a proper value,
  {                                   the program continues execution.
     printf("input = '%s'\n", instring);
     preblank(instring);
     tblank(instring);
     printf("output = '%s'\n", instring);
  }
  else                           <--- Otherwise the program prints an
  {                                   error message and stops running.
     printf("You didn't enter a character string.\n");
  }
}
```

Having made the corrections to *tblank.c* and *main.c*, try running the program through *lint* again:

```
host% make lint
lint  main.c  preblank.c  adjstring.c  tblank.c
main.c:
preblank.c:
adjstring.c:
tblank.c:
```

Lint does not display any error messages.

Now rebuild the program, and test it by entering **Ctrl-d**:

```
host% make
cc  -c  main.c
cc  -c  tblank.c
cc  main.o  preblank.o  adjstring.o  tblank.o  -o  blankstrip
host% blankstrip
Enter a string with leading and trailing blanks:
Ctrl-d
You didn't enter a character string.
```

The program appears to work properly now. Well, almost. If you enter a string with trailing blanks, *blankstrip* still misbehaves:

```
host% blankstrip
Enter a string with leading and trailing blanks:
asdf
input = 'asdf     '
output = 'asdf     '
```

Lint cannot decipher problems with a program's logic.

6.2 For More About *lint* . . .

Look at the manual page for *lint* (type **man lint**). You will find that the utility has many options. You may also want to read the article *Lint, a C Program Checker*. This article can be found in the *UNIX Programmer's Supplementary Documents, Volume 1*, available in the documentation racks in most Athena clusters.

7. The *dbx* Source Code Debugger

The *dbx* source code debugger is one of the most powerful tools available to programmers because it helps you locate **runtime** errors in your source code. Runtime errors, in contrast to **compiletime** errors, are errors in a program's logic, not errors in its syntax. Runtime errors only show up when you execute a program.

Runtime errors come in several flavors:

- Your program runs all right but gives you undesired results.
- Your program does nothing. It may be waiting for you to enter data, or it may be stuck in an infinite loop. If it is infinite looping, the only way you can stop the program is by typing Ctrl-C.
- Your program displays output at a furious and unending rate. This usually indicates an infinite loop and the only way you can get it to stop is by typing Ctrl-C.
- Your program produces a fatal error and generates a **core dumped** error message. This happens when a program violates the operating system's rules by trying something like dividing by zero or accessing memory that doesn't belong to it. When a program errs in this manner, Unix takes a snapshot of its representation in memory and deposits the image in a file named **core** in the current directory. (Note that a *core* file usually takes up a lot of disk space, and unless you plan to use it to help debug the program, you should remove it by typing *rm core*.)

dbx is great for isolating these kinds of problems. Even if you know just a little bit about the debugger, it will take a lot of the pain out of the debugging process.

NOTE: as nice as *dbx* is, the debugger is not perfect. Unfortunately, the debugger has several runtime bugs of its own. See the following *dbx* tutorial for more information. Also see Appendix G for an introduction to the *Saber-C* programmming environment. *Saber-C* provides a more stable debugging environment.

7.1 Using *dbx*

Before you can use *dbx* on a program, you **must** recompile the program using *cc's* **-g** option:

`host%` **cc -g source.c**

Under this option, *cc* creates additional information about the program and deposits it in a **symbol table**. The debugger must have this symbol table to do its work.

You start *dbx* by typing:

`host%` **dbx** ***executefile***

where *executefile* is the executable version of the program. (If you have used *cc* without the *-o* option, *executefile* is *a.out*). As it starts up, *dbx* displays a message similar to the following:

```
dbx version 3.11 of 10/2/85 18:33 (mit-paris)
type 'help' for help.
reading symbolic information ...
```

before displaying the *dbx* command prompt:

```
(dbx)
```

7.2 A Note About *core* Files

If your current directory has a *core* file in it, *dbx* will attempt to use it even if the file has nothing to do with the program you are tying to debug. If *dbx* finds a *core* file, it displays this message before issuing its command prompt:

```
[using memory image in core]
```

Unless you are planning to use a *core* file in debugging, you should remove it with the Unix *rm* command. If you *are* using a *core* file to debug, make sure to remove it when you finish.

7.3 The *dbx* Debugging Process

dbx helps you debug a program by letting you control the program's execution. The debugger lets you set **breakpoints**, places in the program where execution is suspended. With execution suspended, you can use *dbx* to examine the values of the program's variables and expressions. The following is an outline of a simple debugging session:

1. You start *dbx*.
2. You set breakpoints.
3. You have *dbx* run the program until it encounters the first breakpoint.
4. You examine the values of variables and expressions to see if they meet your expectations.
5. You tell *dbx* to continue execution to the next breakpoint, and so on.
6. When you have found the problem, you exit the debugger.

7.4 *dbx* Commands

The following is a list of *dbx's* most useful commands; the others are documented in the *dbx man* page (type **man dbx**). You will see these commands in action in the next section.

quit This command ends your *dbx* session and gets you back to the shell. You can suspend the session by typing Ctrl-Z; restart it by typing *fg*.

help This command prints a summary of *dbx* commands.

run This command tells *dbx* to run your program. The program will run to completion unless it produces a fatal error, or unless you have told *dbx* to halt execution by setting breakpoints with the **stop** command. If your program takes command line arguments, add the arguments to the *run* command.

stop This command tells *dbx* to halt execution when the program reaches a specified line, a specified function, or generates a specified condition. If you want to debug a program by suspending execution and stepping through it line by line, you must use this command to set at least one breakpoint. You can use *stop* as follows:

stop at *sourceline*
: Stop at line number *line* of the source code.

stop in *function*
: Stop when *function* is called.

stop *variable*
: Stop when the value of *variable* changes.

next After *dbx* has suspended execution, you can use this command to execute the next line of source code. If the next line is a function call, *dbx* will execute the entire function as though it were one line, and halt execution at the next line of the original procedure.

step Like *next*, this command executes the next line of source code, but *step* differs from *next* in the following way: if the next line of source code is a function call, *step* will execute the first line of source *within the function*, and will suspend execution at the function's next line of source. You will want to use *step* in cases where you want to examine what happens inside specific function calls.

cont This command will restart execution. The program will run to completion unless it produces a fatal error, or it reaches a breakpoint.

print After *dbx* has stopped execution, you can use this command to display the value of a variable or expression:

print *variable*

print *expression*

You can use *print* to display any valid C expression.

dump This command displays the names and values of all currently active variables.

trace This command displays tracing information as *dbx* executes a program:

trace *variable*

displays *variable's* name and value each time the program changes *variable's* value.

trace *function*

displays information about *function* each time it is called. *dbx* will tell you which routine made the call, the source line number of the call, what parameters were passed to the function, and what value the function returned upon completion.

trace *expression* **at** *sourceline*

displays the value of the *expression* as the program executes the statement at *sourceline*.

trace *sourceline*

displays the statement at *sourceline* before the program executes it. Note that sometimes *trace* doesn't work properly. If you try to *trace* an array variable in a *main*() procedure, *dbx* itself generates a fatal error and produces its own *core* file.

status This command lists by number the currently active *trace* and *stop* commands.

delete This command cancels *trace* and *stop* commands by command number as displayed with the *status* command.

file This is an important command if you are debugging a program made up of multiple source files. When used without arguments, *file* prints the filename of the current source file. When you first start the debugger the current source file is the file that was specified first when you compiled the program. If you compile the program like this:

cc -g main.c preblank.c

main.c will be the current source file when you start *dbx*. You can use *file* to switch current source files as follows:

```
file preblank.c
```

This command will switch the current file to *preblank.c*.

list This command displays source code. If you use it without arguments, *list* will display the next ten lines of source code in the current source file. You can use arguments to *list* to specify lines of code:

list 10, 30

displays source code lines ten through thirty. You can use *list* to display a function's source code, but only if that function is part of the current source file. For instance:

list main

displays the *main*() function's source code if *main* is part of the current source file. If the function is contained in another source file, use the *file* command to switch source files before using *list* to display the code.

where This command is used when you are using *dbx* to examine a *core* file. *where* displays the line of code that produced the fatal error leading to the core dump. Will also list all currently active function calls. *where* will also tell you what breakpoint you're at if you happen to forget.

assign This command lets you change the value of a program's variable in mid-execution. This command:

```
assign slength = slength - 1
```

subtracts 1 from the initial value of *slength*.

8. A *dbx* Tutorial

In this tutorial you will use *dbx* to debug the *blankstrip* program. You will debug the program twice: first by setting breakpoints and executing the program a line at a time, second by using the debugger's *trace* command to print the values of key variables as the program executes.

As shown previously, when you compile and run *blankstrip*, the program removes leading blanks properly, but fails to remove trailing ones:

```
host% blankstrip
Enter a string with leading and trailing blanks:
   asdf
input = '    asdf    '
output = 'asdf    '
```

Before you can debug the program with *dbx* you must recompile it using the *-g* option of *cc*. The *Makefile* will do this for you if you type:

```
host% make debug
cc  -g  main.c  preblank.c  adjstring.c  tblank.c
main.c:
preblank.c:
adjstring.c:
tblank.c:
```

This command creates a debuggable *a.out* file.

Now start the debugger:

```
host% dbx a.out
dbx version 3.12 of 12/6/85 8:35 (paris).
Type 'help' for help.
reading symbolic information ...
```

8.1 Debugging by Setting Breakpoints

Most new *dbx* users find the utility helpful because it lets them execute their programs a line at a time. Before you can use *dbx* in this manner, you must tell the debugger where to halt the program's initial execution. This is done by setting a *breakpoint*.

Use the debugger's *list* command to display *main.c*'s source:

```
(dbx)list
     1
     2   #include <stdio.h>
     3   #define MAXLEN 1000
     4
     5   main()
     6   {
     7     char instring[MAXLEN];
     8
     9     printf("Enter a string with leading and
                  trailing blanks:\n");
    10     if (gets(instring))
```

When you use *list* without arguments, the command displays the source file's next ten lines of code. Use the command again:

```
(dbx) list
    11       {
    12         printf("input = '%s'\n", instring);
    13         preblank(instring);
    14         tblank(instring);
    15         printf("output = '%s'\n", instring);
    16       }
    17     else
    18       {
    19         printf("You didn't enter a string.\n");
    20       }
```

The program doesn't do anything until the first *printf*() statement. This line of code certainly doesn't generate the bug, so you can set the breakpoint at a line after that. Since the next statement, line 11, is the part of the program that takes the data from the user's terminal, set the breakpoint there. Set it by using the *stop* command as follows:

```
(dbx) stop at 10
[1] stop at "main.c":10
```

Now begin the program's execution with the *run* command:

```
(dbx) run
Enter a string with leading and trailing blanks:
[1] stopped in main at line 10 in file "main.c"
    10     if (gets(instring))
```

Execution halts at the breakpoint. When *dbx* halts execution at a breakpoint, it does so **before executing the code on that line**. Our breakpoint halts execution just before *main*() calls the *gets*() function.

You can now step through the program by using the *next* command:

```
(dbx)next
```

Nothing happens. That's because the program is waiting for you to enter the test string.

```
asdf                    <-- Enter data with four leading and trailing blanks.

stopped in main at line 12 in file "main.c"
   12         printf("input = '%s'\n", instring);
```

next is not the only command you could have used to execute the line. The debugger also provides a *step* command. The difference between the commands is this: if the current line of the program is a function call, *next* will execute the call and stop execution on the next line of the program; the *step* command will halt execution on a source line *within* the function being called.

To demonstrate the difference, use the *step* command as follows:

```
(dbx) step
input = '    asdf    '
stopped in main at line 13 in file "main.c"
   13         preblank(instring);
(dbx) step
stopped in preblank.preblank at line 4 in file "preblank.c"
    4       int n = 0;
```

You have *stepped* your way into the *preblank*() function. If you had continued using *next*, *dbx* would have executed the function call and would have halted execution on the next line of *main.c*.

Now that control has passed to *preblank*(), you can display its source with the *list* command:

```
    5
    6       while (s[n] == ' ' || s[n] == '\t')
    7       {
    8         ++n;
    9       }
   10       if (n > 0)
   11       {
   12         adjstring(s, n);
   13       }
   14       return;
```

You could at this point continue by executing this function line by line, but since *blankstrip* appears to be removing leading blanks correctly (that is, *preblank* seems to be working fine), you want to return control back to *main*(). You can do so by setting a new breakpoint at the end of *preblank*(), then *continuing* execution until the new breakpoint is reached:

```
(dbx) stop at 14          <-- Set a breakpoint at the end of the function.

[6] stop at "preblank.c":14

(dbx) cont                <-- Continue execution to the breakpoint.

[6] stopped in preblank.preblank at line 14 in file "preblank.c"
   13            return;
(dbx) step                <-- Step back to main.c.

stopped in main at line 14 in file "main.c"
   14          tblank(instring);
```

Control has passed back to *main*(). The next line within *main.c* is the call to *tblank*(), the function suspected of being buggy.

Now that you are back in *main.c*, display the value of *instring*:

```
(dbx)print instring
"asdf    "
```

preblank() has removed the leading blanks. What will *tblank*() do? Execute the function call with *next*, and then display *instring*'s value:

```
(dbx)next
stopped in main at line 15 in file "main.c"
   15          printf("output = '%s'\n", instring);
(dbx)print instring
"asdf    "
```

The trailing blanks remain; the bug appears to lie within *tblank.c*. Finish executing the program with the *cont* command:

```
(dbx) cont
output = 'asdf    '

execution completed
```

Now you want to remove the old breakpoints and set a new breakpoint in *tblank.c*. Use the *status* command to display the current breakpoints, then remove them with the *delete* command:

```
(dbx)status
[1] stop at "main.c":10
[3] stop at "preblank.c":14
(dbx)delete 1
(dbx)delete 3
(dbx)status
(dbx)
```

You must now find a suitable place to set the new breakpoint. Before you can *list tblank.c*'s source, you need to tell *dbx* to switch from the *main.c* file to the new source file. Do this with the *file* command:

```
(dbx) file tblank.c
```

Now display the file:

```
(dbx) list
     1
     2   tblank(s)
     3   char *s;
     4   {
     5       int slength;
     6
     7       slength = strlen(s);
     8
     9           while((s[slength] == ' ' || s[slength] == '\t'))
    10           {
```

The function's execution begins at line seven, so set the new breakpoint there:

```
(dbx) stop at 7
[4] stop at "tblank.c":7
```

Now run the program again:

```
(dbx) run
Enter a string with leading and trailing blanks:
  asdf
input = '    asdf    '
[6] stopped in tblank.tblank at line 7 in file "tblank.c"
     7       slength = strlen(s);
```

Use the *next* command to go through the function and the *print* command to display *slength*'s value:

```
(dbx) next
stopped in tblank.tblank at line 9 in file "tblank.c"
     9           while((s[slength] == ' ' || s[slength] == '\t'))
(dbx) print slength
8
(dbx) next
stopped in tblank.tblank at line 13 in file "tblank.c"
    13           s[++slength] = '\0';
```

Wait a minute . . . Execution skipped from line 9 to line 13! What's going on? Display the missing lines:

```
(dbx) list 9, 13
     9           while((s[slength] == ' ' || s[slength] == '\t'))
    10           {
    11               --slength;
    12           }
    13           s[++slength] = '\0';
```

In jumping from line 9 to line 13, execution bypasses the *while* loop. This is the part of the program that removes trailing blanks. Apparently the value of *s[slength]* fails the loop's test. In otherwords, the end of the string is not a blank or a tab character. Display the value of *s[slength]*:

```
(dbx)print s[slength]
'\0'
```

The value is supposed to be a blank space, but instead it is the null character. Display more of *s*'s values:

```
(dbx)  print s[slength - 1]
' '                                   <-- Here's a blank.
(dbx)  print s[slength - 2]
' '                                   <-- And another.
```

Aha! That's the bug. The *strlen*() function at line 7 returns the number of characters in the string. The *tblank*() function uses that value--8 in this case--to set the upper boundary of the string's array. But C arrays are indexed starting from 0, not starting from 1. A string that holds 8 characters is indexed from 0 to 7, not 1 to 8. When *tblank*() tests the value of *s[8]* at the head of the *while* loop, it uses the null character that marks the end of the string. The program should subtract 1 from the value returned by *strlen*() before using it as the upper index boundary. You can test the hypothesis like this: rerun the program to the breakpoint in *tblank.c*; after *strlen*() has returned the value of *slength*, subtract 1 from that value and see if the new *s[slength]* satisfies the *while* test.

```
(dbx) cont                            <-- Finish executing the program.
output = 'asdf    '

execution completed
(dbx)run                              <-- Run the program again.

Enter a string with leading and trailing blanks:
    asdf
input = '    asdf    '
[6] stopped in tblank.tblank at line 7 in file "tblank.c
"
    7       slength = strlen(s);
(dbx)next                             <-- Execute the strlen() call.

stopped in tblank.tblank at line 9 in file "tblank.c"
    9           while((s[slength] == ' ' || s[slength] == '\t'))
```

Now use the *assign* command to subract 1 from *slength*:

```
(dbx)  assign slength = slength - 1
(dbx)  print slength
7
```

Continue executing the program:

```
(dbx) next
stopped in tblank.tblank at line 11 in file "tblank.c"

   11              --slength;           <-- Now the loop executes

(dbx) next
stopped in tblank.tblank at line 12 in file "tblank.c"
   12          }
(dbx) next
stopped in tblank.tblank at line 9 in file "tblank.c"
    9          while((s[slength] == ' ' || s[slength] == '\t'))
(dbx) next
stopped in tblank.tblank at line 11 in file "tblank.c"
   11              --slength;
```

Since *tblank*() appears to be working properly, let the program finish executing and see if it produces the correct results:

```
(dbx) cont
output = 'asdf'                         <-- All right!

execution completed
```

Use the *quit* command to exit the debugger:

```
(dbx) quit
```

8.2 Debugging with the *trace* Command

The *dbx* debugger lets you approach debugging from different angles. This time you will debug the program with the *trace* command.

The method used in debugging with *trace* is different from that used when debugging with breakpoints. Although you can set breakpoints and use *trace* at the same time, *trace* works best when you let large pieces of the program run without halting. With *trace* you tell the debugger to monitor specific variables and functions as the program runs, then you let the program execute the lines of code you want traced. The *trace* method of debugging can be more efficient than the breakpoint method, but unfortunately *dbx* sometimes dies when you try to use the *trace* command.

In the case of *blankstrip*, you are interested in discovering what happens to the character string entered at the the terminal. That string is held in *main.c*'s *instring* variable. Start up the debugger, tell *dbx* to trace the *instring* variable, then run the program:

```
host% dbx a.out
dbx version 3.12 of 12/6/85 8:35 (paris).
Type 'help' for help.
reading symbolic information ...
(dbx) trace instring
[1] trace instring in main.main
(dbx) run
Segmentation fault (core dumped)
host%
```

What happened? The debugger itself just crashed on us, and produced a *core* file. Unfortunately, *dbx* can't *trace* array variables in *main*() procedures without dying. You can however, use *trace* to monitor array variables in subfunctions. In the case of our program, you will be able to *trace* the *s* variable in *tblank.c*.

Before you restart the debugger, you should remove the *core* file that *dbx* produced when it crashed:

```
host% rm core
```

If you don't remove *core*, *dbx* will load the file when you restart the debugger. Since the *core* file has little to do with your program (it is an image of a *dbx* crash), it is meaningless to the task at hand.

Start *dbx* again, tell the debugger to trace the string array in *tblank*, then run the program:

```
host% dbx a.out
dbx version 3.12 of 12/6/85 8:35 (paris).
Type 'help' for help.
reading symbolic information ...
(dbx) trace s in tblank
[using tblank.tblank.s]
[1] trace tblank.tblank.s in tblank.tblank
(dbx)run
Enter a string with leading and trailing blanks:
  asdf
input = '    asdf    '
initially (at line 7 in "tblank.c"):    tblank.tblank.s
= "asdf    "
output = 'asdf    '

execution completed
```

In this instance, *trace* tells you that *s*--actually *tblank.tblank.s*--is initially set to *"asdf "*. It doesn't give you any more information because the string does not get modified.

Try running the program again, and this time *trace* the *slength* variable as well:

```
(dbx) trace slength in tblank
[2] trace slength in tblank.tblank
(dbx) run
Enter a string with leading and trailing blanks:
    asdf
input = '    asdf    '
initially (at line 7 in "tblank.c"):    tblank.tblank.s
= "asdf    "
initially (at line 7 in "tblank.c"):    slength = 4
after line 9 in "tblank.c":     slength = 8
after line 14 in "tblank.c":    slength = 9
output = 'asdf    '

execution completed
(dbx) quit
```

This time *trace* displays some important information. It shows that *slength* never becomes smaller, indicating that *tblank* skips the *while* loop. Knowing this information, you can now set a breakpoint in *tblank*() and use *print* to display the elements of the *s* array as you did in the previous debugging session.

8.3 Fixing the Bug

You have determined that the bug sits on line 7 of *tblank.c*:

```
slength = strlen(s);
```

Because of C's array indexing rules (arrays begin with 0, not with 1), *strlen*() returns a value that needs to be decremented before being assigned to *slength*. Edit *tblank.c* so that line 7 looks like this:

```
slength = strlen(s) - 1;
```

Now *make debug*, and try *dbx* again:

```
host% make debug
cc -g main.c preblank.c adjstring.c tblank.c
main.c:
preblank.c:
adjstring.c:
tblank.c:
host% dbx a.out
dbx version 3.12 of 12/6/85 8:35 (paris).
Type 'help' for help.
reading symbolic information ...
```

Try retracing the variables in *tblank.c*:

```
(dbx) trace s in tblank
[using tblank.tblank.s]
[1] trace tblank.tblank.s in tblank
(dbx) trace slength in tblank
[2] trace slength in tblank
(dbx) run
Enter a string with leading and trailing blanks:
  asdf
input = '    asdf    '
initially (at line 7 in "tblank.c"):   tblank.tblank.s
 = "asdf    "
initially (at line 7 in "tblank.c"):   slength = 4
after line 9 in "tblank.c":    slength = 7
after line 12 in "tblank.c":   slength = 6
after line 12 in "tblank.c":   slength = 5
after line 12 in "tblank.c":   slength = 4
after line 12 in "tblank.c":   slength = 3
after line 14 in "tblank.c":   slength = 4
output = 'asdf'

execution completed
(dbx) quit
```

Now that you have repaired the bug, build a new *blankstrip*:

```
host% make
cc  -c  tblank.c
cc  main.o  preblank.o  adjstring.o  tblank.o  -o  blankstrip
```

8.4 Using *dbx* with a *core* File

In this section, you will debug a version of *blankstrip* that produces a fatal error and a *core* dump when it executes. Programs generate fatal errors when they attempt operations that violate the rules of the operating system. This happens when a program tries to divide by zero, or access memory that doesn't belong to it. When a program errs in this manner, Unix takes a snapshot of the program's image in memory and deposits the results into a *core* file.

NOTE: if you are working on an Athena workstation, the tutorial described in this section may not work as described. Unfortunately the workstation *dbx* differs significantly from the timesharing *dbx*. Currently the workstation *dbx* is not as stable as its timesharing counterpart.

You will recompile *blankstrip* using a different version of *main.c*, *maincore.c*. This new version of *main*() differs from the old in one respect; the declararation of the *instring* variable. This variable receives the input string when *main*() calls the *gets*() function. The original *main.c* file declared *instring* like this:

```
char instring[MAXLEN];
```

maincore.c declares it like this:

```
char   *instring;
```

According to the laws of C, both declarations are valid. In fact, if you look at the manual page for *gets*() (type **man 3 gets**), the documentation suggests that you declare the *gets*() argument as a pointer to *char*:

```
char *gets(s)
char *s;
```

The manual page does not suggest that you declare the argument as a character array:

```
     char s[VALUE];
```

Unfortunately the documentation assumes that you understand that ***s** has to point to allocated space. If you don't understand this, and you declare a *char* pointer variable for *gets*() in the wrong context, you create what can be an ugly bug.

Compile and run the program as follows:

```
host% cc -g maincore.c preblank.c adjstring.c tblank.c
maincore.c:
preblank.c:
adjstring.c:
tblank.c:
host% a.out
Enter a string with leading and trailing blanks:
    asdf
Bus error (core dumped)
```

The program generates a fatal error by attempting something contrary to the rules of the operating system. When a program errs in this manner, Unix takes a snapshot of the program in memory, and dumps the results into a *core* file.

You can use *dbx* to examine *core* files, as the following will illustrate:

```
host% dbx a.out
dbx version 3.12 of 12/6/85 8:35 (paris).
Type 'help' for help.
reading symbolic information ...
[using memory image in core]            <-- dbx loads the core file.
```

The *dbx* **where** command tells you what procedures and functions are active. When used to debug *core* files, *where* tells you what procedures were active when the program generated the fatal error:

```
(dbx) where
gets(0x0) at 0x1f3
main(0x1, 0x7fffe45c, 0x7fffe464), line 10 in "maincore.c"
```

The numbers that *dbx* displays refer to addresses in memory. They won't be of much use to you unless you know something about *core* images.

In addition to giving you memory addresses, the message indicates that *gets*() was active when the program produced the fatal error, and that the error happened at line 10 of *maincore.c*. Set a breakpoint there and run the program:

```
(dbx) stop at 10
[1] stop at "maincore.c":10
(dbx) run
Enter a string with leading and trailing blanks:
[1] stopped in main at line 10 in file "maincore.c"
   10     if (gets(instring))
```

At this point *gets*() has not supplied a value for *instring*. Examine the *instring* variable:

```
(dbx) print instring
(nil)
(dbx) print instring[0]
reference through nil pointer
```

The messages show that *instring* is an uninitialized pointer, a pointer that points to an arbitrary space. If you try to assign data to *instring*, you ask the operating system to store the data at whatever address that *instring* addresses.

In this instance, *instring* addresses an invalid space, so the operating system responds by terminating the program and producing the *core* file. Luck was on your side this time. Many times uninitialized pointers point to valid memory locations. When that happens, the program continues running. If the program tries to use the value stored at the address, you may end up with a very obscure runtime error.

Compare the two *instring* declarations again:

```
char instring[MAXLEN];

char *instring;
```

The first declaration says "create an array of type *char*, of size MAXLEN - 1" (MAXLEN - 1 because C arrays are indexed starting from 0). The second declaration says "create a pointer that points to type *char*, but don't bother assigning it any space just yet."

Before you can use a *char* pointer in a program, you must make it point to an array that has already been allocated. Before you use a ***s**, make sure it references something like *instring[MAXLEN]*.

9. Other Sources of Information

This document provides a **brief** introduction to Unix programming tools. If you want to do a lot of programming on Unix, you should look at these other sources of information.

9.1 Unix Manual Pages of Note

To the inexperienced (and sometimes even the very experienced), the Unix manual pages are incomprehensible. Unfortunately, many of them contain information that cannot be found elsewhere. You should know about the following pages:

cc(1)	The C compiler has options for producing symbol tables, output files, and diagnostics. Type **man 1 cc**.
ld(1)	The link editor manual page is required reading. It describes the order in which object files and library archives are processed. Type **man 1 ld**.
make(1)	This manual page covers the basics and lists several useful options. Type **man 1 make**.
lint(1)	The *lint* utility has options for different kinds of diagnostics. Type **man 1 lint**.
dbx(1)	This manual page lists all of the debugger's commands. Type **man 1 dbx**.
a.out(5)	This manual page describes the format of *a.out* files, and describes how the system sets up memory when it executes an *a.out*. The page is difficult reading but you should make the effort. Type **man 5 a.out**.
nm(1)	You can use *nm* to look at the contents of object file symbol tables. It can be a useful debugging tool, especially if you're having linking and loading problems. Type **man 1 nm**.
ar(1), *ranlib(1)*	These manual pages tell you how to create and process your own library archive files. Also see Appendix F of this document for more information. Type **man 1 ar** and **man 1 ranlib**.
lorder(1), *tsort(1)*	These utilities can be used to reorder functions within personal libraries so that *ld* can find all external references in one pass. Type **man 1 lorder** and **man 1 tsort**.
strip(1)	This utility removes symbol tables from executable files, thus making executables smaller. Although *cc* has an option to do just this, you can use *strip* to process an executable without having to recompile the program. Type **man 1 strip**.
strings(1)	This utility lets you look for ascii text within object and executable files. Type **man 1 strings**.
core(5)	This manual page describes the format of *core* files. Type **man 5**

core.

adb(1) — This is another debugger, the debugger of choice for many a Unix wizard, especially for debugging *core* files. Good luck learning how to use it. Type **man 1 adb**.

od(1) — You can use this utility to examine the innards of executable, object, and *core* files. Type **man 1 od**.

9.2 Other Unix Documentation

In addition to the manual pages, The Unix manuals contain many articles and tutorials. You may want to look at some of the ones listed here. The following documents can be found in the *UNIX Programmer's Supplementary Documents, Volume 1*, which in turn can be found in the documentation racks at most Athena clusters. You can also purchase them at the Information Services Publications office, in Building 11, Room 209.

The C Programming Language - Reference Manual is the official statement of the syntax of C. It's pretty dry material.

Lint, a C Program Checker covers everything you ever wanted to know about *lint*, and then some. As a general introduction to the utility, this document is more useful than the *lint(1)* manual page.

A Tutorial Introduction to ADB. Just what it says, an introduction to the wizard's debugger.

Debugging with dbx is another tutorial look at the *dbx* utility. It is a brief document.

Make - A Program for Maintaining Computer Programs should be read by anyone who wants to use *make* on a regular basis.

An Introduction to the Revision Control System introduces **RCS**, a tool that helps groups of programmers work together without stepping on one another's toes. If you are working on a cooperative project, or on a program that needs constant revising, read this document.

9.3 *The UNIX Programming Environment*

This book, written by Brian Kernighan and Rob Pike, describes the AT&T Unix environment, but is still useful to Berkeley Unix programmers. It discusses many useful topics:

- The Unix file system.
- Shell programming.
- Programming with standard I/O.
- Unix system calls.
- Program development.

You can find the book in the documentation racks at most Athena clusters.

A. Preprocessor Options to the *cc* Command

A.1 The *-E* Option to *cc*

The **-E** option to the *cc* command lets you watch the macro preprocessor at work, without actually compiling the program. Process the sample program's *main.c* file like this:

```
host% cc -E main.c
```

The preprocessor responds by displaying the following:

```
# 1 "main.c"

# 1 "/usr/include/stdio.h"

extern  struct  _iobuf {
        int     _cnt;
        char    *_ptr;
        char    *_base;
        int     _bufsiz;          <-- stdio.h inserted
        short   _flag;
        char    _file;
} _iob[];

struct _iobuf   *fopen();
struct _iobuf   *fdopen();
struct _iobuf   *freopen();
struct _iobuf   *popen();
long    ftell();
long    fseek();
char    *fgets();
char    *gets();

char    *sprintf();
# 3 "main.c"


main()
{
  char instring[1000];            <-- MAXLEN replaced by 1000

  printf("Enter a string with leading and trailing blanks\n");
  gets(instring);
  printf("input = '%s'\n", instring);
  preblank(instring);
  trailblank(instring);
  printf("input = '%s'\n", instring);
}
```

This option is useful in situations when you think preprocessor statements are causing mistakes in the rest of your program. Consider the following example. You have written two source files that use the same symbolic constant:

```
*/ aprog.c */

#define NUMBER 7
main()
{
  int num;

  num = NUMBER * NUMBER;
  printf("%d\n", num);
   bprog();
}

/* bprog.c */
bprog()
{
  int num;

  num = NUMBER + NUMBER;
  printf("%d\n", num);
}
```

When you compile the two together, you get an error message:

```
host%  cc aprog.c bprog.c
aprog.c:
bprog.c:
"bprog.c", line 5: NUMBER undefined
```

That's funny. The *#define* statement in *aprog.c* is supposed to set NUMBER to the value 7. Let's see what the preprocessor really does with the *#define* statement:

```
host% cc -E aprog.c bprog.c
aprog.c:
# 1 "aprog.c"


main()
{
  int num;

  num = 7 * 7;                    <-- NUMBER has been changed to 7
  printf("%d\n", num);
   bprog();
}
bprog.c:
# 1 "bprog.c"
bprog()
{
  int num;

  num = NUMBER + NUMBER;          <-- NUMBER has not been changed here!
  printf("%d\n", num);
}
```

As you may have guessed, or already known, a source file's preprocessor statements affect only that source file. The *#define* statement in *aprog.c* does not set the NUMBER constant in *bprog.c*.

A.2 Specifying Other *include* Directories with *-I*

In the course of developing C programs that require their own header files, it sometimes makes sense to keep the header files in a subdirectory so that you don't clutter up the source file directory. When you compile the source files, you can instruct the preprocessor to search the subdirectory by using the *-I* option to *cc*. The option works like this:

host% cc -I*path* *source.c*

where *path* specifies the pathname of the directory that contains the header file you want to include. *Do not* insert a space between the *-I* and the pathname! The *-I* option tells the preprocessor to search the source file's directory first, then the directory specified by *-I*, and finally the system's header file directory.

Suppose you want to compile the *bridge.c* program in your *~username/bridge* directory, and you want to include header files kept in your *~username/bridge/include* directory. Use *cc* as follows:

host% cc -Iinclude bridge.c

You can use *-I* more than once in a command line if you want the preprocessor to search more than one directory.

B. Using Functions in the C Libraries

The Unix system contains several **libraries** of C functions. A **library** is a file that holds a collection of precompiled functions. The Unix documentation classifies the functions as follows:

- The *straight* library functions, also known as the *standard C* functions. Included among these are string-manipulation functions such as *strcat*() and *strcpy*(), and memory-allocation functions such as *malloc*() and *realloc*(). The C compiler has *ld* load this library automatically at compile time.
- The standard I/O library functions. These include the *printf*() and *scanf*() functions, as well as file I/O functions such as *fopen*() and *fclose*(). These functions are loaded automatically at compile time. See the *INTRO(3S)* manual page for a list of these functions (type **man 3s intro**).
- The math library functions. Included among these are the trigonometric functions, the logarithm functions, and the bessel functions. These functions **are not** loaded automatically at compile time. To access these functions, you must use *cc*'s *-l* option. For more information, see Appendix E. See the *INTRO(3M)* manual page for a list of these functions (type **man 3m intro**).
- The internet network library functions. These functions are automatically loaded at compile time. See the *INTRO(3n)* manual page for a list of these functions (type **man 3n intro**).
- Miscellaneous function libraries such as the *curses* library and the *ndbm* library. *Curses* is a screen updating and cursor movement optimization library; *ndbm* is a database library. The miscellaneous libraries are not loaded automatically at compile time. To access them, you must use *cc*'s *-l* option. See the *INTRO(3X)* manual page for a list of these libraries (type **man 3x intro**).

The *straight* functions, the standard I/O functions, the network functions, and some of the miscellaneous functions are kept in the */lib/libc.a* file. This file is sometimes referred to as the *libc* library, and the *standard C run-time* library. When you use *cc* to compile your program, the link editor *ld* automatically searches this file for missing functions.

Note that *ld* does not automatically search all the system's libraries. The most notable among those libraries not searched is the math library, and others include the *curses* library and the *ndbm* library. If the program you're compiling calls functions that are not members of the *libc* library, you must explicity link to the other libraries at compile time. This operation is described below.

B.1 Unix Manual Pages for the Library Functions

In previous versions of Athena's Unix documentation, you could find the library function manual pages in Section 3 of the *Unix Functions Manual*. These manual pages were grouped together library by library, with the *straight* functions documented in Section 3, the standard I/O functions in Section 3s, the math functions in Section 3m, and so on.

The UNIX 4.3 documentation has placed the function manual pages within Section 3 of the *UNIX Programmer's Reference Manual*. But the functions are no longer grouped together by library. Section 3 now documents the functions in alphabetical order, without regard to function type.

B.2 Linking to Other Libraries: the *-l* Option to *cc*

When you compile programs that need access to libraries that are not automatically searched, you must explicitly link to those libraries by using the *-l* option to *cc*. (Strictly speaking, *-l* is not an option to *cc*, it is an option to the link editor *ld*; *cc* passes the option on to *ld* during the loading and linking phase of compilation.)

Use the option as follows:

host% cc *file1.c file2.c* . . . . -l*abbrev*

where *abbrev* is the abbreviated name of the library you want to search. Do not insert a space between the *-l* and the abbreviation.

How do you find out a library's abbreviation? Take the math library for instance. The math functions are kept in the file */usr/lib/libm.a*. Unix convention suggests that a system library filename begin with a *lib* prefix and end with a *.a* suffix, *lib*m*.a* for example. If you remove the prefix and suffix from the filename, you are left with the library's abbreviation. The math library's abbreviation is *m*.

So you link to the math library as follows:

host% cc arithmetic.c trigonometry.c calculus.c -lm

You can use *-l* more than once on the same command line. Sometimes you have to. Suppose you wanted to compile a program that used fuctions from the *curses* library. Certain *curses* routines call functions from the *termcap* library, so you have to call both libraries at compile time. The *curses* library is contained in */usr/lib/libcurses.a*, the *termcap* library in */usr/lib/libtermcap.a*. You link to the libraries as follows:

host% cc screen.c display.c chortle.c -lcurses -ltermcap

When you specify a library with the *-l* option, *ld* will search for the library in the standard directories */lib*, */usr/lib*, and *usr/athena/lib*.

C. Some Useful Functions from the Standard Library

The functions and macros described below are part of the *lib/libc.a* library. The link editor *ld* automatically searches this library at compile time. Note that the functions described here reperesent only a partial list of the functions contained in *libc.a*. See the *intro(3)* manual page for a complete list (type **man 3 intro**).

The functions listed below are described very briefly. Please read the appropriate manual pages before using the functions in your own work. You will save yourself a lot of grief.

C.1 ASCII Conversion Functions

These functions convert ASCII character strings to numbers.

```
atoi()          convert ascii string to an integer

atof()          convert ascii string to a floating point

atol()          convert ascii string to a long integer
```

These functions are documented on the *atof(3)* manual page. Type **man 3 atof** for more information.

Many programmers use these functions to convert character data entered via the *gets*() function into numerical representation as demonstrated in the following test program:

```
/* This program tests the gets() and atof() functions */
#include <stdio.h>
#define SIZE 50     /* Set arbitrary limit for array size */
main()
{
    double atof();    /* Must declare the atof() function */
    char input_num[SIZE];  /* Holds input ascii digits */
    double float_num;    /*  Holds converted value */

    printf("Enter a real (floating point) number\n");
    gets(input_num);     /* Store user's response in input_num */
    printf("%s\n", input_num);   /* Print input_num */
    float_num = atof(input_num); /* Convert input_num to float */
    printf("%f\n", float_num);   /* Print the float value */
}
```

C.2 Testing for ASCII Characters: the *ctype* Macros

The macros listed below test the nature of character data. They test data one character at a time. The *isalpha* macro tests for letters, the *isdigit* macro tests for digits, and so on.

The macros are defined in the standard header file *ctype.h*, so you must include this file if you want to use the macros:

```
#include <ctype.h>
```

Here is a list of the macros:

```
isalpha          isdigit
isupper          ispunct
islower          isprint
isalnum          iscntrl
isspace          isascii
```

They are all documented on the *ctype(3)* manual page. For more information, type **man 3 ctype**.

C.3 Functions that Manipulate Strings

The following functions help you examine and process string data. You can automatically declare the functions by including the standard header file *strings.h* into your program:

```
#include <strings.h>
```

Here's a list of the string functions:

strlen(s)	**find the number of non-null characters in the string** *s***.**
strcmp(s1, s2)	**compare string** *s1* **to** *s2***.**
strncmp(s1, s2, n)	**compare** *n* **characters of** *s1* **to** *n* **characters of** *s2***.**
strcat(s1, s2)	**append a copy of string** *s2* **to the end of string** *s1***.**
strncat(s1, s2, n)	**append at most** *n* **characters of** *s2* **to the end of** *s1***.**
strcpy(s1, s2)	**copy string** *s2* **to string** *s1***.**
strncpy(s1, s2, n)	**copy** *n* **characters of** *s2* **to** *s1***.**
index(s, c)	**point to first occurrence of character** *c* **in string** *s***.**
rindex(s, c)	**point to last occurrence of character** *c* **in string** *s***.**

These functions are documented on the *string(3)* manual page. For more information type **man 3 string**.

D. The Standard I/O Library

The following is a partial list of the functions in the standard I/O library. For a complete list, see the *intro(3S)* manual page. Type **man 3s intro**. This manual page is an indispensable introduction to I/O programming in Unix.

The descriptions given below are very brief. Please consult the appropriate manual pages before trying to use these functions in your work.

D.1 The *stdio.h* Header File

Many of these functions rely on definitions contained within the system header file *stdio.h*. You should include *stdio.h* in any source file that uses the standard I/O functions:

```
#include <stdio.h>
```

D.2 Character String Input

gets(s)	**get string** *s* **from standard input.**
fgets(s, n, stream)	**read** *n* **- 1 characters from** *stream* **and store in string** *s*.

These functions are documented on the *gets(3)* manual page. Type **man 3s gets** for more information.

D.3 Character and Word Input

getchar()	**gets the next character from standard input.** *getchar*() **is actually a macro defined in** *stdio.h*.
getc(stream)	**gets the next character from input** *stream*. *getc*() **is a macro defined in** *stdio.h*.
fgetc(stream)	**behaves like** *getc*()**, but is an actual function.**
getw(stream)	**gets the next word from input** *stream*.

These functions are documented on the *getc(3S)* manual page. Type **man 3s getc** for more information.

D.4 Formatted Output

`printf(format [, args])`	**print output as specified by** *format* **to the standard output stream.**
`fprintf(stream, format [,args])`	**print output as specified by** *format* **to** *stream***.**
`sprintf(s, format [, args])`	**place output on string** *s***.**

These functions are documented on the *printf(3S)* manual page. Type **man 3s printf** for more information.

D.5 Character and Word Output

`putchar(c)`	**print character** *c* **on standard output.** *putchar* **is a macro defined in** *stdio.h***.**
`putc(c, stream)`	**print character** *c* **on** *stream***.** *putc* **is a macro defined in** *stdio.h***.**
`fputc(c, stream)`	**behaves like** *putc* **but is a real function.**
`putw(w, stream)`	**appends word** *w* **to output** *stream***.**

These functions are documented on the *putc(3S)* manual page. Type **man 3s putc** for more information.

D.6 String Output

`puts(s)`	**copy string** *s* **to standard output.**
`fputs(s, stream)`	**copy string** *s* **to output** *stream***.**

These functions are documented on the *puts(3S)* manual page. Type **man 3s puts** for more information.

D.7 Formatted Input

scanf(format [, p_args])	**read characters from standard input, and interpret them according to** *format*. **Store input in variables pointed to by** *p_args*.
fscanf(stream, format [, p_args])	**same as** *scanf*() **but read from input** *stream*.
sscanf(s, format [, p_args])	**same as** *scanf*(), **but read from string** *s*.

These functions are documented on the *scanf(3S)* manual page. Type **man 3s scanf** for more information.

E. The Math Library and its Functions

E.1 The Math Library */usr/lib/libm.a*

This library contains all of C's standard math functions, with the exception of the *abs*() (absolute value) function and the random number functions. *abs*() is considered one of C's *straight* functions; you will find it in the standard function library */lib/libc.a* and you can read its documentation by typing **man 3 abs**. The random number generating functions are also part of the standard function library. You can read their documentation by typing **man random**.

The math library is one of those libraries that the C compiler does not automatically search when it compiles your program. If your program uses math functions, you need to explicitly link to */usr/lib/libm.a*, as follows:

```
host% cc source.c -lm
```

The descriptions given below are very brief. Please read the appropriate manual pages before using these functions in your work.

E.2 The *math.h* Header File

Every function in the math library is declared in the system's standard *math.h* header file (/usr/include/math.h). Include this header file into every source file that makes calls to the math library:

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

E.3 Math Function Arguments are *Doubles*

Every function in the math library returns a *double precision* value, and expects its arguments to be of type *double* as well. If you fail to supply *double* typed arguments, your program could have runtime problems.

E.4 The Trigonometric Functions

```
sin(var)    asin(var)        atan2(var)
cos(var)    acos(var)
tan(var)    atan(var)
```

The trigonometric functions are documented on the *sin(3M)* manual page. Type **man 3m sin** for more information.

E.5 The Hyperbolic Functions

```
sinh(var)
cosh(var)
tanh(var)
```

The hyperbolic functions are documented on the *sinh(3M)* manual page. Type **man 3m sinh** for more information.

E.6 Square Root, Exponential, Power, and Log Functions

```
sqrt(var)                      log(var)
exp(var)                       log10(var)
pow(basevar, powervar)
```

These functions are documented on the *exp(3M)* manual page. Type **man 3m exp** for more information.

E.7 The Bessel Functions

```
j0(var)     y0(var)
j1(var)     y1(var)
jn(var)     yn(var)
```

These functions are documented on the *j0(3M)* manual page (that's a zero). Type **man 3m J0** for more information.

E.8 Euclidean Distance Functions

```
cabs(z)          hypot(x,y)
```

These functions are documented on the *hypot(3M)* manual page. Type **man 3m hypot** for more information.

E.9 The Log Gamma Function

```
gamma(x)
```

This function is documented of the *gamma(3M)* manual page. Type *man 3m hypot* for more information.

E.10 Ceiling, Floor, and Absolute Value Functions

```
ceil(x)          fabs(x)
floor(x)         rint(x)
```

These functions are documented on the *floor(3M)* manual page. Type **man 3m floor** for more information.

F. Creating Your Own Libraries: *ar* and *ranlib*

You can create your own function libraries. Many of Athena's Faculty Projects have done so. After you have written and tested the functions, you create the library using the following procedure:

1. compile the functions into object files with the *-c* option to *cc*;
2. use the **ar** (archive) command to condense the functions into one library file; and
3. use the **ranlib** command to create the library's table of contents.

Suppose that you have written the following functions to help you with math-related programs:

```
/* factorial() calculates the factorial of its argument
   and returns the result */
factorial(integer)
int integer;
{
        int n, answer;

        for (n = answer = 1; n <= integer; ++n)
        {
                answer *= n;
        }

        return(answer);   /* Return the factorial of integer */
}
```

and

```
#include <math.h>
/* sq() squares its argument and returns the answer */
double sq(operand)
double operand; /* must be double for pow() */
{
        double answer;
        double exponent = 2.00; /* must be double for pow() */

        answer = pow(operand, exponent);
        return(answer);
}
```

You now want to create a library *mymath.a* in which to archive these functions (the **.a** extension is a Unix convention; it tells the world that the file is a function *archive*, or library). This is what you do:

1. Compile the functions:

   `host%` cc -c factorial.c sq.c

2. Use the *ar* command to create the library *mymath.a* and insert the function objects.

   `host%` ar q mymath.a factorial.o sq.o

   The *q* tells *ar* to "quickly" add the functions to the list of functions in the library. This means that if the library hasn't been created yet, *ar* will create it, then insert the functions.

3. Finish by using the *ranlib* command to create a **table of contents** for the library:

   `host%` ranlib mymath.a

   ranlib creates the table of contents and inserts it into *mymath.a*. The table is very important; the link editor *ld* needs it during the linking and loading phase to search the library for missing functions.

Once you have created a library, you can use the *ar* utility to add new functions or delete old ones as occasion demands. Remember to process your personal libraries with *ranlib* each time you change something. For more information on *ar*, type:

`host%` man ar

For more information on *ranlib*, type:

`host%` man ranlib

F.1 Linking to Personal Libraries

You can link to a personal library at compile time by adding the pathname of the library to the *cc* command as follows:

`host%` cc source.c *librarypath*

Where *librarypath* specifies the library file's pathname.

Suppose you had a program called *factest.c* that tested the *factorial* function in your *mymath.a* library. Compile as follows:

`host%` cc factest.c mymath.a

Say that you keep all your personal libraries in a *lib* subdirectory beneath your home directory. You would compile the program like this:

`host%` cc factest.c ~/lib/mymath.a

Note: you do not use *cc*'s *-l* option to link to personal libraries.

The library's pathname must come after the source's filename. You can link to as many personal libraries as you like. If functions in one library depend on functions in another, beware of the order you use to link them.

F.2 Questions of Order when Linking to Multiple Libraries

The order in which you link to libraries when compiling can be extremely important if those libraries happen to have functions that call one another. Take a look at the *sq*() function once again:

```
#include <math.h>
double sq(operand)
double operand; /* must be double for pow() */
{
        double answer;
        double exponent = 2.00; /* must be double for pow() */

        answer = pow(operand, exponent);
        return(answer);
}
```

The function calls the standard math function *pow*() to perform the actual squaring operation, *sq*()won't work unless it is linked to the math library.

Suppose you have a program *sqtest.c* that calls *sq*(). Assume that the function is part of the *mymath.a* library. If you compile *sqtest.c* like this:

```
host% cc sqtest.c -lm mymath.a
```

the link editor *ld* will respond with this error message:

```
Undefined:
_pow
```

If you compile *sqtest.c* like this:

```
host% cc sqtest.c mymath.a -lm
```

the program compiles properly.

Why the difference? When *ld* searches libraries, it does so in the order that they are listed on the *cc* command line. It searches each library **once**, and only loads those functions that resolve external references.

In the first *sqtest.c* example, *ld* searched the system math library first, then the *maymath.a* library. Consequently, the *sq*() function got loaded after *ld* searched the math library. The link editor did not load *pow*() from the math library because it didn't know that *sq*() would need *pow*(). When *ld* did load *sq*() from *mymath.a*, it could not resolve call for *pow*(). Hence the error message.

In the second example, *ld* searched *mymath.a* first, and loaded *sq*(). Then *ld* searched the math library, and loaded *pow*() because it knew that the function an external reference to *sq*(). *sq*()'s call to *pow*() was resolved and the program compiled properly.

So the rule of thumb is this: if you need to link to libraries that make calls to one another, link the calling functions **before** the functions being called.

G. The Saber-C Programming System

Saber-C is an interactive programming system for C programmers. It has a lot of features, only a few of which will be introduced in this Appendix. Here we'll concentrate on using *Saber-C* to debug source code.

As a source level debugger, *Saber-C* provides many of the features that come with *dbx*. You can list source code with it, set breakpoints, step through functions a line at a time, and trace a program's execution.

Saber-C also has features that *dbx* doesn't provide. Because *Saber* can be used as a C language interpreter, you can debug a program or any of its separate functions without having to compile. You load source files into *Saber-C*, run them in a controlled fashion to look for bugs, then reload edited source files, and continue testing. Best of all, *Saber* provides a stable debugging environment. It won't arbitrarily crash on you like *dbx*.

The system has been successfully integrated into the X window environment. When you use *Saber*'s on-line help system, the utility creates a separate help window to display the text. When you run a program inside the *Saber* environment, *Saber* creates a program input/output window.

Because *Saber-C* is a C language interpreter, all commands issued in the environment must follw the rules of C syntax. Every command statement must end with a semi-colon; every function must be called with parentheses; etc.

In fact, every *Saber* command is a C function. Suppose for instance that you want to access *Saber*'s on-line help system. Use the **help()** function as follows:

```
3 -> help();
```

See the final Section of this Appendix for more information about the on-line help system and other documentation for the *Saber-C* programming environment.

G.1 Debugging the *blankstrip* Program

What follows is an abbreviated version of the debugging session presented in Chapter 8, this time performed with the *Saber C* system. Assume that the *tblank*() function still contains the bug.

G.1.1 Starting Saber-C

Start *Saber-C* as follows:

```
host% saber
```

The debugger displays a version/license message, then asks if you want instructions. If you respond to the instructions prompt by pressing **Return**, **Enter**, or **y**, *Saber* creates a *help* window on your workstation's screen and lets you page through several windows worth of introductory text. If you respond by entering a **n**, *Saber* forgoes the *help* window.

In any case, the debugger attaches the standard C library and displays the *Saber-C* prompt:

```
Creating help window ...
Attaching: /lib/libc.a              <-- Saber attaches the library,
1 ->                                <-- then displays the Saber prompt.
```

G.1.2 Loading Source Files

Use *Saber*'s **load()** function to load *blankstrip*'s source files:

```
1 -> load("main.c");
Loading: main.c
(int) 0                             <-- load() returns int 0 to indicate success.
2 -> load("adjstring.c");
Loading: adjstring.c
(int) 0
3 -> load("preblank.c");
Loading: preblank.c
(int) 0
4 -> load("tblank.c");
Loading: tblank.c
(int) 0
5 ->
```

You can also load files when you start *Saber*:

```
host% saber main.c adjstring.c preblank.c tblank.c
```

G.1.3 Running blankstrip within Saber-C

You execute programs within the *Saber* environment with *Saber*'s **run()** function:

```
5 -> run();
Creating program input/output window ...
Executing: a.out
```

The *run*() function creates a *program input/output* window to handle the program's data input and output. Move the mouse cursor into the input/output window and enter a test string with leading and trailing blanks. After you enter the test string, the program removes the leading blanks, but fails to remove the trailing ones.

As the program runs, *Saber* displays the following in the original window:

```
Linking from /lib/libc.a ...Linking completed.
Linking from /lib/libc.a ...Linking completed.
Program exiting with return status = 0.
Resetting to top level.
6 ->
```

G.1.4 Setting a Breakpoint

Use *Saber*'s **stop()** function to set a breakpoint in the *tblank*() function:

```
6 -> stop(tblank);
Breakpoint #1 set at "tblank.c", line 7, tblank().
(int) 1
7 ->
```

When you provide the *stop* function with a function name, *Saber* sets the breakpoint at that function's first line of *executable* code. In this instance, the breakpoint is set at *tblank*()'s line 7.

Now execute the program with *run*(), and enter the test string in the input/output window. The original window displays something like the following:

```
7 -> run();

executing: a.out
Stopped in tblank(s = (char *) 0xd3008 "asdf    ") at "tblank.c", line 7
    6:
*B  7:   slength = strlen(s);
    8:
(break 1) 8 ->
```

Saber halts execution at this line of code before executing the line.

Use the **list()** function to display *tblank*()'s source code:

```
(break 1) 8 -> list();
*B  7:   slength = strlen(s);
    8:
    9:   while((s[slength] == ' ' || s[slength] == '\t'))
   10:     {
   11:       --slength;
   12:     }
   13:   s[++slength] = '\0';
   14:   return;
   15: }
   16:
More:  <space>/<ret>/back/quit/edit ?
```

The *list*() function is similar to the Unix *more* command. Enter a *space* to see the next page of text, use the *Return* or *Enter* key to see the next line of text, etc. Enter a **q** to quit the listing.

G.1.5 Stepping through tblank

Now use the **step()** function to execute *tblank*() a line at a time:

```
(break 1) 9 -> step();
Stopped in tblank() at "tblank.c", line 9
*   9:   while((s[slength] == ' ' || s[slength] == '\t'))
(break 1) 10 -> step();
Stopped in tblank() at "tblank.c", line 13
*  13:   s[++slength] = '\0';     <-- Execution skips from line 9 to 13!
```

The *tblank* function doesn't work because execution bypasses the *while* loop responsible for stripping the trailing blanks. Look at the value of *s[slength]* to see if it is a *space* or *tab* character:

```
(break 1) 11 -> s[slength];
(char) '\x000'                    <-- It isn't a space or a tab . . .
```

You know that the data you gave to program contained trailing blanks, so where are they? Look at *s[slength - 1]*:

```
(break 1) 12 -> s[slength - 1];
            (char) ' '            <-- There's a space character.
(break 1) 13 -> s[slength - 2];
            (char) ' '            <-- And another.
```

Use the *step*() function to finish stepping through *tblank*(), then use the **cont()** function to finish executing the program:

```
(break 1) 14 -> step();
Stopped in tblank() at "tblank.c", line 14
*  14:   return;
(break 1) 15 -> cont();
Program exiting with return status = 0.
Resetting to top level.
16 ->
```

G.1.6 Fixing the Bug Midflight

Apparently the *tblank*() function sets the variable *slength* improperly. Line 7 in *tblank*() says this:

```
slength  =  strlen(s);
```

when it should say this:

```
slength = strlen(s) - 1;
```

Test the hypothesis by running the program again and resetting the *slength* variable:

```
16 -> run();
Executing: a.out
Stopped in tblank(s = (char *) 0xd3008 "asdf    ") at "tblank.c", line 7
    6:
*B  7:   slength = strlen(s);
    8:
(break 1) 17 -> step();
Stopped in tblank() at "tblank.c", line 9
*   9:   while((s[slength] == ' ' || s[slength] == '\t'))
(break 1) 18 -> s[slength];           <-- What is s[slength]?
(char) '\x000'
(break 1) 19 -> s[slength - 1];
(char) ' '
(break 1) 20 -> slength = slength - 1;       <-- Reset the variable.
(int) 7
(break 1) 21 -> s[slength];
(char) ' '                            <-- Now s[slength] is a space character.
```

Step through the function and see if this fixes the bug:

```
(break 1) 22 -> step();
Stopped in tblank() at "tblank.c", line 11
*  11:       --slength;               <-- tblank() now executes the loop.
(break 1) 23 -> step();
Stopped in tblank() at "tblank.c", line 9
*   9:   while((s[slength] == ' ' || s[slength] == '\t'))
(break 1) 24 ->
```

Use the *cont*() function to finish executing the program:

```
(break 1) 24 -> cont();
Program exiting with return status = 0.
Resetting to top level.
25 ->
```

This time the program returns a string with both leading and trailing blanks stripped. Use the **quit()** function to exit *Saber-C*.

```
25 -> quit();
host%
```

G.2 Where to go from Here

As mentioned earlier, *Saber-C* has an extensive on-line help facility. Use the **help()** function as follows to get introduced to the help system:

```
3 -> help("intro");
4 -> help("hints");
```

The *help*() function displays a nice summary of all *Saber* command if you call the function without arguments:

```
7 -> help();
```

The *help*() function will provide detailed information on individual commands if you call the function using the command name as an argument:

```
9 -> help(list);
10 -> help(stop);
11 -> help(trace);
```

You can get a more comprehensive tour of the *Saber* system by looking at the document "A Quick Tour of Saber-C," which is part of the *SABER User's Guide*.