Some Fundamentals (a review)

As a primer, we always have a lecture on the following topics for CS302 students:

a) .h files
b) the meaning of the extern keyword
c) compiling versus linking
d) how a process's memory is organized (stack, heap, etc)
e) makefiles

Let's start!

What is a program?

Well, depends on who you talk to, you will get very different answers. Of course, in operating systems, when a program is running, we all know it's called a 'process'. The program itself can be in many forms, i.e. machine code, assembly code, C code, C++ or Fortran, Java, …

Processors execute machine code ONLY. Machine code is written with binary machine words composed of op codes and operands. The op codes represent different instructions, for instance an arithmetic addition. Operands are operated on by the instruction. A machine word may look like:

10001000110101001010101101101011

In this case, **'10001000'** encodes the op code, `110101001010` and `101101101011` are the two operands. Well, most sane people don't want to write code in this format. Naturally, our forerunners sought a way out, using mnemonics to represent all those binary streams. Then the line above may become:

```
ADD AX, BX
```

These mnemonics and some other directive commands are collectively called assembly language. The assembly program now looks more readable and maintainable:

```
0000000000000100    MOV BX, VALUE1
0000000000001000    MOV AX, FACTOR
0000000000001100    MUL AX, BX
```

(By the way, this is how CPU goes about to scale VALUE1 by FACTOR. VALUE1 x FACTOR doesn't make any sense to the processor). '04', '08' and '0C', written in HEX numbers, are memory address where corresponding lines in the program are stored.

Well, life became easier, but not easy enough. If one had no other obligations in life, he could write a lot of efficient programs just using these commands. But if one has a life away from a keyboard, there are problems:

1. The mnemonics in assembly language are in one-to-one correspondence with machine codes, which are processor dependent. If you move from Intel x86 to Motorola 68k, a new program has to be written. Here is a real assembly program that runs on 68k:

```
          ORG    $1000
N         EQU    5
CNT1      DC.B   0
CNT2      DC.B   0
ARRAY     DC.B   2,7,1,6,3

          ORG    $1500
MAIN      LEA    ARRAY,A2
          MOVE.B #N,D1
          CLR    D6
          CLR    D7
          JSR    SORT
          STOP   #$2700

SORT      MOVE.B #0,D6
          MOVE.B #1,D7
LOOP      MOVE.B $0(A2,D6.W),D2
          MOVE.B $0(A2,D7.W),D3
          CMP.B  D2,D3
          BGT    EXCHANGE
          ADD.B #1,D7
          CMP.B #5,D7
          BLT    LOOP
          JMP    CHECK1
EXCHANGE MOVE.B D2,$0(A2,D7.W)
          MOVE.B D3,(A2,D6.W)
          ADD.B #1,D7
          CMP.B #5,D7
          BLS   LOOP
CHECK1   ADD.B #1,D6
          MOVE.B D6,D7
          CMP.B #4,D6
          BLT   LOOP
          RTS
```

```
END    $1500
```

By the way, if you want to see how your C program would look like wearing Assembly, try using "cc –S xxx.c", or "gcc –S xxx.c". These commands produce .s files, which contains the assembly version of your C code. Take a close look and get a feeling of how different they are on Suns and Linux.

Note, although assembly code are very close to what machine code is, they are still different. A tool called "assembler" converts assembly code into true machine code (the realm of binary!).

2. In the above program, the addresses are physical addresses, which correspond to individual bytes on your memory banks. If you have hard coded these addresses, you better only run one program on your processor at each time! NOTE, really, only one program can be run.
   This works fine for small scale embedded systems that are almost bare machines (have you ever built one yourself?). Unfortunately, as soon as you have a need for operating systems, this doesn't work anymore. Later, efforts were made in assemblers to allow assembly program to use logical address that can be remapped. That made operating systems possible.

3. How tedious it is to program in assembly language! It may take 30 lines of code to just put a character on your console, provided that you are not making system calls. As an anecdote, while WordPerfect and Word look very similar, one is written in assembly and the other in C.

4. Every time when you want to reuse a piece of code of yours, …... Imagine, using other people's code.

Realizing all those problems, a lot of great scientists got to work. A revolution then gave us compilers, linkers, operating systems, and high level programming languages, such as C/C++, PASCAL, … Let's focus on C, compiler and linker for now.

With C language, we all know that programming became simple. If you were born at the right time, you don't have to be very smart to start making big money with 3 month's night school. This is how you do it:

1. you write a program in certain editor. Let's call your program bigmoney.c.
2. then you run a command "cc bigmoney.c", this command gives an executable named a.out.
3. you type "a.out" at the command prompt, then bigmoney runs.

Unfortunately, as a computer science major at UT, you need to know what is going on in the background. Here is an early look:

1. a ***compiler*** is run and translates your C program into binary machine code and forms a module, which is usually referred to as the 'object'. Inside each object that the compiler spits out, essentially, you get a machine code module in a platform dependent format. For instance, Common Object File Format (COFF). However, there is a difference, no physical addresses encoded in there. This makes the module re-locatable in main memory. Your object, in this case, called bigmoney.o, is written to disk.
   No matter you write programs in C or Assembly language, the files you edited appear as a text file. Someone has to convert all those texts into binary commands. This is the job of compiler and assembler, respectively.

2. a ***separate*** linker program is then invoked to look through your object file for any function calls or variables that are unresolved (unresolved external references). For instance, you called "`printf("Hello World\n");`" in your bigmoney.c. Your compiler didn't know how that call works because

it's provided in the system library. It is your linker's job to dig a machine code module of `printf` out of system library and put it together with your bigmoney.o. The result of your linking session is an executable file on disk.

3. after you type in a.out at the command prompt, the ***loader*** in operating system reads a.out from disk, and put it in main memory. It is at this time the physical address of every line in a.out is determined. Then, the operating system finds the entry point of your program (the main function in bigmoney.c) and start executing from there.

Let's start answering those questions at the top now.

Why do we need .h file?

It takes several persons to develop any serious application using libraries in the operating system, and sometimes, libraries provided by other people. Modular development is the norm. Compilers need to perform type checks and make sure your calls to the library and other peoples' functions are correct. No matter you have access to the C source of the function body that you are calling or not, it is a bad idea to paste a function body into your code (why?). Using .h files, which usually incorporates data types, function declarations and macros, resolves this issue. As a kind suggestion, you don't want to paste a function definition into a .h file, either. (why?)

Compiling versus Linking

Compiler converts each source file into an object file. Linker takes all generated object file, as well as the system libraries that are relevant, and builds an executable file that is stored on disk. Actually, it is possible to link object modules created from code written in different languages (C/C++/Fortran/Assembly) together, as long as the linkage specification (protocol) is known and abided by.

How is a process's memory organized?

Let' s take a deeper look at a program. After an operating system loads a.out from disk into main memory, in order for a.out to run, there are several segments that must exist:

Executable text
: Consists of the actual executable instructions in the binary. This is mapped from the on-disk binary and is read-only.

Executable data
: Contains initialized variables present in the executable. This is mapped from the on-disk binary, but has permissions set to read/write/private (the private mapping ensures that changes to these variables during runtime are not reflected in the on-disk file, or in other processes sharing the same executable).

Heap space
: Consists of memory allocated by means of `malloc(3)`. This is called anonymous memory, because it has no mapping in the filesystem.

Stack

Also allocated from anonymous memory.

The exact layout of the memory address space of a process is system dependent. However, there is a naming convention of different segments that almost every system follows. Let's take BSD Unix as an example.

BSD defines segments to be contiguous regions of virtual space (like everybody else). A process address space is composed of five primary segments: (i) the *text* segment, holding the executable code; (ii) the *initialized data* segment, containing those data that are initialized to specific non-zero values at process start-up; (iii) the *bss* segment, containing data initialized as zero at process start-up; (iv) the *heap* segment, containing uninitialized data and the process's heap; and (v) the *stac*k.

Beyond the stack is a region holding *the kernel's stack* (used when executing system calls on behalf of this process, for example) and the *user struct*, a kernel data structure holding a large quantity of process-specific information.

This is a illustration of BSD's per process virtual memory space. Text, initialized data, bss and heap are contiguously put in memory address, starting from the lowest address. Stack, however, starts from the highest address space, expanding downward in address space. The empty 'chunk' between the top of heap and bottom of stack are allocated as needed.

Function and the Stack segment

Talking about function, one has to mention the term 'lifetime'. Lifetime discusses the aspect of a variable during the time when a program is running. A global variable is always in existence from the program is started by the OS till the program exits and is cleared from memory. A variable declared inside a function, however, is only extant during the time when the function is being called. This is due to the fashion in which a function call is implemented.

1. the OS pushes the current address in main program onto stack
2. pushes all necessary data to recover running the main program, including CPU registers, onto stack
3. pushes all arguments in that function call onto stack one-by-one
4. finds the head address of the text segment of that function and start executing
5. inside the function, allocated more memory space in the stack for local variables
6. if there are malloc() calls, allocate memory from heap space
7. finish computing in function
8. if had dynamic memory allocations, maybe you need to deallocate (well-behaving) ?
9. pop all local variables off stack
10. pop all function arguments off stack

11. recover original running state from stack
12. start executing again from the address just got from stack

Question: how is the return value of a function passed back to the main program?

Scope and Visibility

An identifier's "visibility" determines the portions of the program in which it can be referenced—its "scope." An identifier is visible (i.e., can be used) only in portions of a program encompassed by its "scope," which may be limited (in order of increasing restrictiveness) to the file, function, block, or function prototype in which it appears. The scope of an identifier is the part of the program in which the name can be used. This is sometimes called "lexical scope." There are four kinds of scope: function, file, block, and function prototype.

All identifiers except labels have their scope determined by the level at which the declaration occurs. The following rules for each kind of scope govern the visibility of identifiers within a program:

File scope
: The declarator or type specifier for an identifier with file scope appears outside any block or list of parameters and is accessible from any place in the translation unit after its declaration. Identifier names with file scope are often called "global" or "external." The scope of a global identifier begins at the point of its definition or declaration and terminates at the end of the translation unit.

Function scope
: A label is the only kind of identifier that has function scope. A label is declared implicitly by its use in a statement. Label names must be unique within a function.

Block scope
: The declarator or type specifier for an identifier with block scope appears inside a block or within the list of formal parameter declarations in a function definition. It is visible only from the point of its declaration or definition to the end of the block containing its declaration or definition. Its scope is limited to that block and to any blocks nested in that block and ends at the curly brace that closes the associated block. Such identifiers are sometimes called "local variables."

Function-prototype scope
: The declarator or type specifier for an identifier with function-prototype scope appears within the list of parameter declarations in a function prototype (not part of the function declaration). Its scope terminates at the end of the function declarator.

However, variables and functions declared at the external level with the **static** storage-class specifier are visible only within the source file in which they are defined. All other functions are globally visible.

extern and static In C

All external variables and functions are of **extern** type of default. External variables are variables at file scope. Variable declarations at the external level are either definitions of variables ("defining declarations"), or references to variables defined elsewhere ("referencing declarations").

An external variable declaration that also initializes the variable (implicitly or explicitly) is a defining declaration of the variable. A definition at the external level can take several forms:

- A variable declared with the **static**. You can explicitly initialize the **static** variable with a constant expression. If you omit the initializer, the variable is initialized to 0 by default.

```
static int k = 16;
static int k;
```

- A variable explicitly initialized at the external level. For example, `int j = 3;` is a definition of the variable `j`.

Once a variable is defined at the external level, it is visible throughout the rest of the translation unit. The variable is not visible prior to its declaration in the same source file. Also, it is not visible in other source files of the program, unless a referencing declaration makes it visible, as described below.

***static** declarator*:

- When modifying a variable, the **static** keyword specifies that the variable has static duration (it is allocated when the program begins and deallocated when the program ends) and initializes it to 0 unless another value is specified.
- When modifying a variable or function at file scope the **static** keyword specifies that the variable or function has internal linkage (its name is not visible from outside the file in which it is declared).

```
// Example of the static keyword
static int i;              // Variable accessible only from this file
static void func();        // Function accessible only from this file
int max_so_far( int curr )
{
   static int biggest;     // Variable whose value is retained
                           //    between each function call
   if( curr > biggest )
      biggest = curr;
   return biggest;
}
```

- You can define a variable at the external level only once within a program. You can define another **static** variable with the same name in a different translation unit, however, no conflict occurs.
- If you declare a function **static**, its name is invisible outside of the file in which it is declared. A function declared to be **static** is visible only within the source file in which it is defined. Functions in the same source file can call the **static** function, but functions in other source files cannot access it directly by name. You can declare another **static** function with the same name in a different source file without conflict.

***extern** declarator*:

- The **extern** keyword declares a variable or function and specifies that it has external linkage (its name is visible from files other than the one in which it's defined). When modifying a variable, **extern** specifies that the variable has static duration (it is allocated when

the program begins and deallocated when the program ends). The variable or function may be defined in another source file, or later in the same file.

- The **extern** keyword declares a reference to a variable defined elsewhere. You can use an **extern** declaration to make a definition in another source file visible, or to make a variable visible prior to its definition in the same source file. Once you have declared a reference to the variable at the external level, the variable is visible throughout the remainder of the translation unit in which the declared reference occurs.
- For an **extern** reference to be valid, the variable it refers to must be defined once, and only once, at the external level. This definition (without the **extern** storage class) can be in any of the translation units that make up the program.
- Functions declared as **extern** are visible throughout all source files in the program (unless you later redeclare such a function as **static**). Any function can call an **extern** function. Function declarations that omit the storage-class specifier are **extern** by default.

Example

The example below illustrates external declarations:

```
/*************************************************************
                      SOURCE FILE ONE
*************************************************************/

extern int i;                /* Reference to i, defined below */
void next( void );           /* Function prototype            */

void main()
{
    i++;
    printf( "%d\n", i );     /* i equals 4 */
    next();
}

int i = 3;                   /* Definition of i */

void next( void )
{
    i++;
    printf( "%d\n", i );     /* i equals 5 */
    other();
}

/*************************************************************
                      SOURCE FILE TWO
*************************************************************/

extern int i;                /* Reference to i in  */
                             /* first source file  */
void other( void )
{
    i++;
```

```
    printf( "%d\n", i );      /* i equals 6 */
}
```

The two source files in this example contain a total of three external declarations of `i`. Only one declaration is a "defining declaration." That declaration,

```
int i = 3;
```

defines the global variable `i` and initializes it with initial value 3. The "referencing" declaration of `i` at the top of the first source file using **extern** makes the global variable visible prior to its defining declaration in the file. The referencing declaration of `i` in the second source file also makes the variable visible in that source file. If a defining instance for a variable is not provided in the translation unit, the compiler assumes there is an

```
extern int x;
```

referencing declaration and that a defining reference

```
int x = 0;
```

appears in another translation unit of the program.

All three functions, `main`, `next`, and `other`, perform the same task: they increase `i` and print it. The values 4, 5, and 6 are printed. If the variable `i` had not been initialized, it would have been set to 0 automatically. In this case, the values 1, 2, and 3 would have been printed.

Lifetime

"Lifetime" is the period during execution of a program in which a variable or function exists. The storage duration of the identifier determines its lifetime, either static duration (global lifetime) or automatic duration (local lifetime).

An identifier declared with the *storage-class-specifier* **static** has static storage duration. Identifiers with static storage duration (also called "global") have storage and a defined value for the duration of a program. Storage is reserved and the identifier's stored value is initialized only once, before program startup. An identifier declared with external or internal linkage also has static storage duration.

An identifier declared without the **static** storage-class specifier has automatic storage duration if it is declared inside a function or block. An identifier with automatic storage duration (a "local identifier") has storage and a defined value only within the block where the identifier is defined or declared. An automatic identifier is allocated new storage each time the program enters that block, and it loses its storage (and its value) when the program exits the block. Identifiers declared in a function with no linkage also have automatic storage duration.

The following rules specify whether an identifier has global (static) or local (automatic) lifetime:

- Global lifetime: All functions have global lifetime. Therefore they exist at all times during program execution. Identifiers declared at the external level (that is, outside all blocks in the program at the same level of function definitions) always have global (static) lifetimes.
- Local lifetime: If a local variable has an initializer, the variable is initialized each time it is created (unless it is declared as **static**). Function parameters also have local lifetime. You can specify global lifetime for an identifier within a block by including the **static** storage-class specifier in its declaration. Once declared **static**, the variable retains its value from one entry of the block to the next.

Although an identifier with a global lifetime exists throughout the execution of the source program (for example, an externally declared variable or a local variable declared with the **static** keyword), it may not be visible in all parts of the program.

Memory can be allocated as needed (dynamic) if created through the use of special library routines such as **malloc**. Since dynamic memory allocation uses library routines, it is not considered part of the language.

Pointers

A pointer is a number, indeed, an integer number representing an address in memory. On the one hand, many programming languages do not have the concept of pointer. However, one of the biggest advantages of C, that is, flexibility, comes from the use of pointer. On the other hand, array is a concept that most programming languages have. In C, we also have the same thing.

Interestingly, in:

`int myarray [30];`

the variable `myarray` really is a pointer of type `(int *)`. In C, array elements are accessed using pointers:

`myarray[5]` can be translated into `*(myarray + 5)`.

This allows arrays that are dynamically allocated inside a function to be passed out of that function.

Further, we can define pointers to pointers, simply because a pointer is just a number representing address. Let's take another look.

```
int myarray[30];
main()
{
    int * myptr;
    int ** mydblptr;
    myptr = myarray;
    mydblptr = &myarray;
}
```

`myarray` is a variable of pointer type in Init. Data segment. There are 30 integers allocated in bss segment as well. Those 120 bytes in bss are the place where you actually put your 30 integers, and `myarray` equals to the head address of those 120 bytes. Say, `myarray = 0000F0CF`. Now, tell me what are the values of `myptr` and `mydblptr`? Please try the above code?

Provided that we all understand the above, let's try this:

```
void f1(int * ptr, int len)
{
   int i;
   ptr = (int*) malloc (sizeof(int)*len);
   for (i = 0; i < len; i ++)
       ptr[i] = i;
```

```
}

main()
{
   int i, * array;
   f1(array, 5);
   for (i = 0; i < 5; i ++)
       printf("got value %d\n", array[i]);
   return;
}
```

You are lucky to not get a segment fault when you run this code. Why? Can you smart people guess what does 'segment fault' mean? What about 'bus error'?

Talking about "segment fault", it may remind you of "page fault". Actually, page fault is so much easier to explain. A page fault is generated when the OS detects that a process is trying to access a page in the virtual memory address space, but that page is not in the physical memory. As a result of that, the OS stops this process until that requested page is read in. Page fault is, in most cases, not an error.

Segment fault is almost always an error, antithetically, much more complicated to explain, and usually processor dependent. Here is a brief list of possible causes on Pentiums (since I can't find anything about any workstation processor, go open standards!!)

- Exceeding segment limit when using CS, DS, ES, FS, or GS
- Exceeding segment limit when referencing a descriptor table
- Transferring control to a segment that is not executable
- Writing to a read-only data segment or a code segment
- Reading from an execute-only segment
- Loading SS with a read-only descriptor
- Loading SS, DS, ES, FS, or GS with a descriptor of a system segment
- Loading DS, ES, FS, or GS with the descriptor of an executable segment that is also not readable
- Loading SS with the descriptor of an executable segment
- Accessing memory through DS, ES, FS, or GS when the segment register contains a NULL selector

Bus error takes place another step down into hardware. All memory R/Ws need to go through a hardware address decoder to get translated into real voltage signals on address bus. Memory and every I/O device are all mapped in the same global address space. Some section in the global address spaces are reserved, some are protected, and some are empty, and some are open. If the hardware decoder can't convert a legal signal out from the pointer you gave, then, it throws back a hardware interrupt – "Bus Error".

Make

This is a little bit involving a topic. There is a great tutorial from GNU on-line. Please consult that for details. Here is a primer that I made from the GNU Make Manual.

Managing a project with a large number of files is always troublesome. Typing compiler and linker commands at command line is error prune. Make is a utility meant to aid the process of managing, building and updating your grand projects. Make needs to have a set of rules specified such that automatic processing can be achieved. These rules are usually specified in a makefile. By default, when make looks for the makefile, it tries the following names, in order: *makefile* and *Makefile*. Normally you should call your makefile either *makefile* or *Makefile*. If make finds none of these names, it does not use any makefile. If you want to use a nonstandard name for your makefile, you can specify the makefile name with the -f or --file option. The arguments -f *name* or --file=*name* tell make to read the file *name* as the makefile. If you use more than one -f or --file option, you can specify several makefiles. All the makefiles are effectively concatenated in the order specified. The default makefile names *makefile* and *Makefile* are not checked automatically if you specify -f or --file.

What does a Rule Look Like

A simple makefile consists of ``rules" with the following shape:

target ... : *dependencies ...*
command
...
...

A *target* is usually the name of a file that is generated by a program; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as clean.
A *dependency* is a file that is used as input to create the target. A target often depends on several files.
A *command* is an action that make carries out. A rule may have more than one command, each on its own line.

Please note: you need to put a tab character at the beginning of every command line! This is an obscurity that catches the unwary.

Usually a command is in a rule with dependencies and serves to create a target file if any of the dependencies change. However, the rule that specifies commands for the target need not have dependencies. For example, the rule containing the delete command associated with the target clean does not have dependencies. A *rule*, then, explains how and when to remake certain files which are the targets of the particular rule. make carries out the commands on the dependencies to create or update the target. A rule can also explain how and when to carry out an action. A makefile may contain other text besides rules, but a simple makefile need only contain rules. Rules may look somewhat more complicated than shown in this template, but all fit the pattern more or less.

Here is a straightforward makefile that describes the way an executable file called edit depends on eight object files which, in turn, depend on eight C source and three header files. In this example, all the C files include *defs.h*, but only those defining editing commands include *command.h*, and only low level files that change the editor buffer include *buffer.h*.

```
edit : main.o kbd.o command.o display.o \
    insert.o search.o files.o utils.o
```

```
        cc -o edit main.o kbd.o command.o display.o \
                insert.o search.o files.o utils.o

main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
```

We split each long line into two lines using backslash - new line; this is like using one long line, but is easier to read.

To use this makefile to create the executable file called *edit*, type:

```
make
```

To use this makefile to delete the executable file and all the object files from the directory, type:

```
make clean
```

In the example makefile, the targets include the executable file edit, and the object files main.o and kbd.o. The dependencies are files such as main.c and defs.h. In fact, each .o file is both a target and a dependency. Commands include cc -c main.c and cc -c kbd.c.

When a target is a file, it needs to be recompiled or relinked if any of its dependencies change. In addition, any dependencies that are themselves automatically generated should be updated first. In this example, *edit* depends on each of the eight object files; the object file *main.o* depends on the source file *main.c* and on the header file *defs.h*.

A shell command follows each line that contains a target and dependencies. These shell commands say how to update the target file. A tab character must come at the beginning of every command line to distinguish commands lines from other lines in the makefile. (Bear in mind that make does not know anything about how the commands work. It is up to you to supply commands that will update the target file properly. All make does is execute the commands in the rule you have specified when the target file needs to be updated.)

The target clean is not a file, but merely the name of an action. Since you normally do not want to carry out the actions in this rule, clean is not a dependency of any other rule. Consequently, make never does anything with it unless you tell it specifically. Note that this rule not only is not a dependency, it also does not have any dependencies, so the only purpose of the rule is to run the specified commands. Targets that do not refer to files but are just actions are called *phony targets*.

How make Processes a Makefile

By default, make starts with the first rule (not counting rules whose target names start with .). This is called the *default goal*. (*Goals* are the targets that make strives ultimately to update.)

In the simple example of the previous section, the default goal is to update the executable program *edit*; therefore, we put that rule first. Thus, when you give the command:

```
make
```

make reads the makefile in the current directory and begins by processing the first rule. In the example, this rule is for relinking *edit*; but before make can fully process this rule, it must process the rules for the files that *edit* depends on, which in this case are the object files. Each of these files is processed according to its own rule. These rules say to update each .o file by compiling its source file. The recompilation must be done if the source file, or any of the header files named as dependencies, is more recent than the object file, or if the object file does not exist. The other rules are processed because their targets appear as dependencies of the goal. If some other rule is not depended on by the goal (or anything it depends on, etc.), that rule is not processed, unless you tell make to do so (with a command such as make clean).

Before recompiling an object file, make considers updating its dependencies, the source file and header files. This makefile does not specify anything to be done for them---the .c and .h files are not the targets of any rules---so make does nothing for these files. But make would update automatically generated C programs, such as those made by Bison or Yacc, by their own rules at this time. After recompiling whichever object files need it, make decides whether to relink *edit*. This must be done if the file *edit* does not exist, or if any of the object files are newer than it. If an object file was just recompiled, it is now newer than *edit*, so *edit* is relinked. Thus, if we change the file *insert.c* and run make, make will compile that file to update *insert.o*, and then link *edit*. If we change the file *command.h* and run make, make will recompile the object files *kbd.o*, *command.o* and *files.o* and then link the file *edit*.

Variables Make Makefiles Simpler

In our example, we had to list all the object files twice in the rule for *edit* (repeated here):

```
edit : main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
       cc -o edit main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

Such duplication is error-prone; if a new object file is added to the system, we might add it to one list and forget the other. We can eliminate the risk and simplify the makefile by using a variable. *Variables* allow a text string to be defined once and substituted in multiple places later. It is standard practice for every makefile to have a variable named objects, OBJECTS, objs, OBJS, obj, or OBJ which is a list of all object file names. We would define such a variable objects with a line like this in the makefile:

```
objects = main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o
```

Then, each place we want to put a list of the object file names, we can substitute the variable's value by writing $(objects). Here is how the complete simple makefile looks when you use a variable for the object files:

```
objects = main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)
main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit $(objects)
```

Letting make Deduce the Commands

It is not necessary to spell out the commands for compiling the individual C source files, because make can figure them out: it has an *implicit rule* for updating a .o file from a correspondingly named .c file using a cc -c command. For example, it will use the command cc -c main.c -o main.o to compile *main.c* into *main.o*. We can therefore omit the commands from the rules for the object files. When a .c file is used automatically in this way, it is also

automatically added to the list of dependencies. We can therefore omit the .c files from the dependencies, provided we omit the commands. Here is the entire example, with both of these changes, and a variable objects as suggested above:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
edit : $(objects)
       cc -o edit $(objects)
main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
       -rm edit $(objects)
```

When the objects of a makefile are created only by implicit rules, an alternative style of makefile is possible. In this style of makefile, you group entries by their dependencies instead of by their targets. Here is what one looks like:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
edit : $(objects)
       cc -o edit $(objects)
$(objects) : defs.h
kbd.o command.o files.o : command.h
display.o insert.o search.o files.o : buffer.h
```

Here *defs.h* is given as a dependency of all the object files; *command.h* and *buffer.h* are dependencies of the specific object files listed for them.
Whether this is better is a matter of taste: it is more compact, but some people dislike it because they find it clearer to put all the information about each target in one place.

Rules for Cleaning the Directory

Compiling a program is not the only thing you might want to write rules for. Makefiles commonly tell how to do a few other things besides compiling a program: for example, how to delete all the object files and executables so that the directory is clean.
Here is how we could write a make rule for cleaning our example editor:

```
clean:
```

```
rm edit $(objects)
```

In practice, we might want to write the rule in a somewhat more complicated manner to handle unanticipated situations. We would do this:

```
.PHONY : clean
clean :
	-rm edit $(objects)
```

This prevents make from getting confused by an actual file called *clean* and causes it to continue in spite of errors from rm. A rule such as this should not be placed at the beginning of the makefile, because we do not want it to run by default! Thus, in the example makefile, we want the rule for edit, which recompiles the editor, to remain the default goal. Since clean is not a dependency of edit, this rule will not run at all if we give the command make with no arguments. In order to make the rule run, we have to type make clean.

Arguments to Specify the Goals

The *goals* are the targets that make should strive ultimately to update. Other targets are updated as well if they appear as dependencies of goals, or dependencies of dependencies of goals, etc. By default, the goal is the first target in the makefile (not counting targets that start with a period). Therefore, makefiles are usually written so that the first target is for compiling the entire program or programs they describe. If the first rule in the makefile has several targets, only the first target in the rule becomes the default goal, not the whole list. You can specify a different goal or goals with arguments to make. Use the name of the goal as an argument. If you specify several goals, make processes each of them in turn, in the order you name them.

Any target in the makefile may be specified as a goal (unless it starts with - or contains an =, in which case it will be parsed as a switch or variable definition, respectively). Even targets not in the makefile may be specified, if make can find implicit rules that say how to make them.
One use of specifying a goal is if you want to compile only a part of the program, or only one of several programs. Specify as a goal each file that you wish to remake. For example, consider a directory containing several programs, with a makefile that starts like this:

```
.PHONY: all
all: size nm ld ar as
```

If you are working on the program size, you might want to say make size so that only the files of that program are recompiled.

Another use of specifying a goal is to make files that are not normally made. For example, there may be a file of debugging output, or a version of the program that is compiled specially for testing, which has a rule in the makefile but is not a dependency of the default goal.

Another use of specifying a goal is to run the commands associated with a phony target or empty target. Many makefiles contain a phony target named *clean* which deletes everything except source files. Naturally, this is done only if you request it explicitly with make clean. Following is a list of typical phony and empty target names. As an example, here are several standard target names which GNU software packages use.

all
: Make all the top-level targets the makefile knows about.

clean
: Delete all files that are normally created by running make.

install

Copy the executable file into a directory that users typically search for commands; copy any auxiliary files that the executable uses into the directories where it will look for them.

tar

Create a tar file of the source files.

dist

Create a distribution file of the source files. This might be a tar file, or a shar file, or a compressed version of one of the above, or even more than one of the above.

Introduction to Pattern Rules

A pattern rule contains the character % (exactly one of them) in the target; otherwise, it looks exactly like an ordinary rule. The target is a pattern for matching file names; the % matches any nonempty substring, while other characters match only themselves. For example, %.c as a pattern matches any file name that ends in .c. s.%.c as a pattern matches any file name that starts with s., ends in .c and is at least five characters long. (There must be at least one character to match the %.) The substring that the % matches is called the *stem*.

% in a dependency of a pattern rule stands for the same stem that was matched by the % in the target. In order for the pattern rule to apply, its target pattern must match the file name under consideration, and its dependency patterns must name files that exist or can be made. These files become dependencies of the target.
Thus, a rule of the form

```
%.o : %.c ; command...
```

specifies how to make a file *n.o*, with another file *n.c* as its dependency, provided that *n.c* exists or can be made.

There may also be dependencies that do not use %; such a dependency attaches to every file made by this pattern rule. These unvarying dependencies are useful occasionally. A pattern rule need not have any dependencies that contain %, or in fact any dependencies at all. Such a rule is effectively a general wildcard. It provides a way to make any file that matches the target pattern. Pattern rules may have more than one target. Unlike normal rules, this does not act as many different rules with the same dependencies and commands. If a pattern rule has multiple targets, make knows that the rule's commands are responsible for making all of the targets. The commands are executed only once to make all the targets. When searching for a pattern rule to match a target, the target patterns of a rule other than the one that matches the target in need of a rule are incidental: make worries only about giving commands and dependencies to the file presently in question. However, when this file's commands are run, the other targets are marked as having been updated themselves.

Suppose you are writing a pattern rule to compile a .c file into a .o file: how do you write the cc command so that it operates on the right source file name? You cannot write the name in the command, because the name is different each time the implicit rule is applied. What you do is use a special feature of make, the *automatic variables*. These variables have values computed afresh for each rule that is executed, based on the target and dependencies of the rule. In this example, you would use $@ for the object file name and $< for the source file name. Here is a table of automatic variables:

`$@`

The file name of the target of the rule. If the target is an archive member, then `$@` is the name of the archive file. In a pattern rule that has multiple targets, `$@` is the name of whichever target caused the rule's commands to be run.

`$%`

The target member name, when the target is an archive member. For example, if the target is *foo.a(bar.o)* then `$%` is *bar.o* and `$@` is *foo.a*. `$%` is empty when the target is not an archive member.

`$<`

The name of the first dependency. If the target got its commands from an implicit rule, this will be the first dependency added by the implicit rule.

`$?`

The names of all the dependencies that are newer than the target, with spaces between them. For dependencies which are archive members, only the member named is used.

`$^`

The names of all the dependencies, with spaces between them. For dependencies which are archive members, only the member named is used. A target has only one dependency on each other file it depends on, no matter how many times each file is listed as a dependency. So if you list a dependency more than once for a target, the value of `$^` contains just one copy of the name.

`$+`

This is like `$^`, but dependencies listed more than once are duplicated in the order they were listed in the makefile. This is primarily useful for use in linking commands where it is meaningful to repeat library file names in a particular order.

`$*`

The stem with which an implicit rule matches. If the target is *dir/a.foo.b* and the target pattern is *a.%.b* then the stem is *dir/foo*. The stem is useful for constructing names of related files.
In a static pattern rule, the stem is part of the file name that matched the % in the target pattern.
In an explicit rule, there is no stem; so $* cannot be determined in that way. Instead, if the target name ends with a recognized suffix, $* is set to the target name minus the suffix. For example, if the target name is foo.c, then $* is set to foo, since .c is a suffix. GNU make does this bizarre thing only for compatibility with other implementations of make. You should generally avoid using $* except in implicit rules or static pattern rules.
If the target name in an explicit rule does not end with a recognized suffix, $* is set to the empty string for that rule.

$?

is useful even in explicit rules when you wish to operate on only the dependencies that have changed. For example, suppose that an archive named *lib* is supposed to contain copies of several object files. This rule copies just the changed object files into the archive:

```
lib: foo.o bar.o lose.o win.o
    ar r lib $?
```

Here is an example of pattern rules actually predefined in make. This rule compiles all .c files into .o files:

```
%.o : %.c
    $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

defines a rule that can make any file *x.o* from *x.c*. The command uses the automatic variables $@ and $< to substitute the names of the target file and the source file in each case where the rule applies.

Wildcards

Wildcards can be used in the commands of a rule, where they are expanded by the shell. For example, here is a rule to delete all the object files:

```
clean:
	rm -f *.o
```

Wildcards are also useful in the dependencies of a rule. With the following rule in the makefile, `make print` will print all the `.c` files that have changed since the last time you printed them:

```
print: *.c
	lpr -p $?
	touch print
```

This rule uses *print* as an empty target file. The automatic variable `$?` is used to print only those files that have changed.

Note: Wildcard expansion does not happen when you define a variable. Thus, if you write this:

```
objects = *.o
```

then the value of the variable `objects` is the actual string `*.o`. However, if you use the value of `objects` in a target, dependency or command, wildcard expansion will take place at that time. To set `objects` to the expansion, instead use:

```
objects := $(wildcard *.o)
```