

Lesson 12

Objectives: -

- Compiling Multi-File Programs
- Reading the Command Line
-

Compiling Multi-File Programs

This process is rather more involved than compiling a single file program. Imagine a program in three files prog.c, containing main(), func1.c and func2.c. The simplest method of compilation (to produce a runnable file called a.out) would be

```
cc prog.c func1.c func2.c
```

If we wanted to call the runnable file prog we would have to type

```
cc prog.c func1.c func2.c -o prog
```

In these examples, each of the .c files is compiled, and then they are automatically linked together using a program called the loader ld.

Separate Compilation

We can also compile each C file separately using the cc -c option. This produces an object file with the same name, but ending in .o. After this, the object files are linked using the linker. This would require the four following commands for our current example.

```
cc -c prog.c
```

```
cc -c func1.c
```

```
cc -c func2.c
```

```
ld prog.o func1.o func2.o -o prog
```

Each file is compiled before the object files are linked to give a runnable file.

The advantage of separate compilation is that only files which have been edited since the last compilation need to be re-compiled when re-building the program. For very large programs this can save a lot of time.

The make utility is designed to automate this re-building process. It checks the times of modification of files, and selectively re-compiles as required. It is an excellent tool for maintaining multi-file programs. Its use is recommended when building multi-file programs.

Using make with Multi-File Programs

We have already used make to build single file programs. It was really designed to help build large multi-file programs. Its use will be described here.

Make knows about 'dependencies' in program building. For example;

- We can get prog.o by running `cc -c prog.c`.
- This need only be done if prog.c changed more recently than prog.o.

make is usually used with a configuration file called Makefile which describes the structure of the program. This includes the name of the runnable file, and the object files to be linked to create it. Here is a sample Makefile for our current example

```
# Sample Makefile for prog
#
# prog is built from prog.c func1.c func2.c
#

# Object files (Ending in .o,
# these are compiled from .c files by make)
OBJS    = prog.o func1.o func2.o

# Prog is generated from the object files
prog: $(OBJS)
    $(CC) $(CFLAGS) -o prog $(OBJS)
# ^^^ This space must be a TAB.
# Above line is an instruction to link object files
```

This looks cluttered, but ignore the comments (lines starting with #) and there are just 3 lines.

When make is run, Makefile is searched for a list of dependencies. The compiler is involved to create .o files where needed. The link statement is then used to create the runnable file.

make re-builds the whole program with a minimum of re-compilation, and ensures that all parts of the program are up to date. It has many other features, some of which are very complicated.

For a full description of all of these features, look at the manual page for `make` by typing

```
man make
```

Reading the Command Line

We've mentioned several times that a program is rarely useful if it does exactly the same thing every time you run it. Another way of giving a program some variable input to work on is by invoking it with *command line arguments*.

(We should probably admit that command line user interfaces are a bit old-fashioned, and currently somewhat out of favor. If you've used Unix or MS-DOS, you know what a command line is, but if your experience is confined to the Macintosh or Microsoft Windows or some other Graphical User Interface, you may never have seen a command line. In fact, if you're learning C on a Mac or under Windows, it can be tricky to give your program a command line at all. Think C for the Macintosh provides a way; I'm not sure about other compilers. If your compilation environment doesn't provide an easy way of simulating an old-fashioned command line, you may skip this chapter.)

C's model of the command line is that it consists of a sequence of words, typically separated by whitespace. Your main program can receive these words as an array of strings, one word per string. In fact, the C run-time startup code is always willing to pass you this array, and all you have to do to receive it is to declare `main` as accepting two parameters, like this:

```
int main(int argc, char *argv[])
{
    ...
}
```

When `main` is called, `argc` will be a count of the number of command-line arguments, and `argv` will be an array ("vector") of the arguments themselves. Since each word is a string which is represented as a pointer-to-char, `argv` is an array-of-pointers-to-char. Since we are not *defining* the `argv` array, but merely declaring a parameter which references an array somewhere else (namely, in `main`'s caller, the run-time startup code), we do not have to supply an array dimension for `argv`. (Actually, since functions never receive arrays as parameters in C, `argv` can also be thought of as a pointer-to-pointer-to-char, or `char **`. But multidimensional arrays and pointers to pointers can be confusing, and we

haven't covered them, so we'll talk about `argv` as if it were an array.) (Also, there's nothing magic about the names `argc` and `argv`. You can give `main`'s two parameters any names you like, as long as they have the appropriate types. The names `argc` and `argv` are traditional.)

The first program to write when playing with `argc` and `argv` is one which simply prints its arguments:

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

```
main(int argc, char *argv[])
{
    int i;

    for(i = 0; i < argc; i++)
        printf("arg %d: %s\n", i, argv[i]);
    return 0;
}
```

(This program is essentially the Unix or MS-DOS `echo` command.)

If you run this program, you'll discover that the set of "words" making up the command line includes the command you typed to invoke your program (that is, the name of your program). In other words, `argv[0]` typically points to the name of your program, and `argv[1]` is the first argument.

There are no hard-and-fast rules for how a program should interpret its command line. There is one set of conventions for Unix, another for MS-DOS, another for VMS. Typically you'll loop over the arguments, perhaps treating some as option flags and others as actual arguments (input files, etc.), interpreting or acting on each one. Since each argument is a string, you'll have to use `strcmp` or the like to match arguments against any patterns you might be looking for. Remember that `argc` contains the number of words on the command line, and that `argv[0]` is the command name, so if `argc` is 1, there are no arguments to inspect. (You'll never want to look at `argv[i]`, for $i \geq \text{argc}$, because it will be a null or invalid pointer.)

As another example, also illustrating `fopen` and the file I/O techniques of the previous chapter, here is a program which copies one or more input files to its standard output. Since "standard output" is usually the screen by default, this is therefore a useful program for displaying files. (It's analogous to the obscurely-named Unix `cat` command, and to the MS-DOS `type` command.) You might also want to compare this program to the character-copying program of section 6.2.

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

```
main(int argc, char *argv[])
{
```

```

int i;
FILE *fp;
int c;

for(i = 1; i < argc; i++)
    {
    fp = fopen(argv[i], "r");
    if(fp == NULL)
        {
        fprintf(stderr, "cat: can't open %s\n", argv[i]);
        continue;
        }

    while((c = getc(fp)) != EOF)
        putchar(c);

    fclose(fp);
    }

return 0;
}

```

As a historical note, the Unix cat program is so named because it can be used to **concatenate** two files together, like this:

```
cat a b > c
```

This illustrates why it's a good idea to print error messages to stderr, so that they don't get redirected. The "can't open file" message in this example also includes the name of the program as well as the name of the file.

Yet another piece of information which it's usually appropriate to include in error messages is the reason why the operation failed, if known. For operating system problems, such as inability to open a file, a code indicating the error is often stored in the global variable `errno`. The standard library function `strerror` will convert an `errno` value to a human-readable error message string. Therefore, an even more informative error message printout would be

```

fp = fopen(argv[i], "r");
if(fp == NULL)
    fprintf(stderr, "cat: can't open %s: %s\n",
            argv[i], strerror(errno));

```

If you use code like this, you can `#include <errno.h>` to get the declaration for `errno`, and `<string.h>` to get the declaration for `strerror()`.

Types of errors:

Semantic Errors

Semantic errors in the grammar can be trivial typos or the sign of a bad design of the FD structure. In any case check the following points:

- Misspellings: wrong feature-name, wrong leaf-value-name
- Paths: in both forms, *i.e.*, ((f1 ((f2 ((f3 and {f1 f2 f3. Make sure that paths actually point to the location in the total FD that you expect.
- The most common source of error is to include the wrong number of up-arrows ^ in a path expression. Always double-check your relative paths, especially those appearing embedded in ALTs, CONTROL, EXTERNAL when the up-arrow notation can be quite counter-intuitive.
- Observe the structure of the unified graph before unification. To this end, use the function uni-fd and pretty-print its output as in:
- (pprint (clean-fd (uni-fd input :grammar gr)))

This is often instructive. Use the function top-gdp to explore this output FD in detail.

- Draw a graph of the total FD with all features instantiated to help you check all your paths and levels of embedding.
- If you do not understand the current structure of the FD, insert a %break% in your grammar at some critical point, and use the function path-value to inspect the total FD during the time of the unification.
- It is risky to use absolute paths in general (SURGE does not include a single absolute path for example). Using an absolute path is a sign of desperation.

Programming Errors

It is quite likely that your attempt to write a *hello world* program will be completely successful and will work first time. It will probably be the only C program you ever write that works first time. There are many different types of mistake you might make when writing C programs. These can result in messages from the compiler, or sometimes the linker, programs that bring the computer grinding to a halt or programs that simply produce the wrong answer to whatever problem they were trying to solve or programs that get stuck in a never-ending loop. As a beginner you are more likely to encounter compiler or linker error messages, particularly if you have already done some programming in another programming language. The rest of this section illustrates some possible errors and their consequences.

In the first example the programmer has forgotten that the C programming language is case sensitive.

```
MAIN()
{
    printf("hello, world\n");
}
```

This error gave rise to some rather mysterious error messages from the linker. Typical examples are shown below. The compiler on the IBM 6150 used to write these notes produced the following error output.

```
ld: Undefined -
    .main
    _main
    _end
```

The compiler on a SUN Sparc Station produced the following messages.

```
ld: Undefined symbol
    _main
Compilation failed
```

The Turbo C integrated environment produced the following.

```
Linker Error: Undefined symbol _main in function main
```

And finally Microsoft C version 5.1 produced the following error messages.

```
hw.c

Microsoft (R) Overlay Linker Version 3.65
Copyright (C) Microsoft Corp 1983-1988. All rights reserved.

Object Modules [.OBJ]: HW.OBJ
Run File [HW.EXE]: HW.EXE /NOI
List File [NUL.MAP]: NUL
Libraries [.LIB]: SLIBCEC.LIB

LINK : error L2029: Unresolved externals:

_main in file(s):
C:\MSC\LIB\SLIBCEC.LIB(dos\crt0.asm)

There was 1 error detected
```

All these errors reflect the fact that the linker cannot put the program together properly. On Unix based systems the linker is usually called "ld". Along with the user supplied main() function all C programs include something often called the **run-time support package** which is actually the code that the operating system kicks into life when starting up your program. The run-time support package then expects to call the user supplied function main(), if there is no user supplied main() then the linker cannot finish the installation of the run-time support

package. In this case the user had supplied MAIN() rather than main(). "MAIN" is a perfectly valid C function name but it isn't "main".

The rather extensive set of messages from Microsoft C 5.1 were partly generated by the compiler, which translated the program without difficulty, and partly generated by the linker. The reference to "crt0" is, in fact, a reference to the C Run Time package which tries to call main() to start the program running.

In the second example the programmer, probably confusing C with another programming language, had used single quotes rather than double quotes to enclose the string passed to printf().

```
main()
{
    printf('hello, world\n');
}
```

On the IBM 6150 the compiler produced the following error messages. The reference to "hw.c" is to the name of the source file that was being compiled.

```
"hw.c", line 3: too many characters in character constant
"hw.c", line 3: warning: Character constant contains more than one byte
```

The SUN Sparc Station compiler gave the following error messages.

```
"hw.c", line 3: too many characters in character constant
Compilation failed
```

The Turbo Integrated environment gave the following messages.
C:\ISPTESTS\HW.C was the name of the source file.

```
Error C:\ISPTESTS\HW.C 3:
Character constant too long in function main
```

Microsoft C version 5.1 gave the following messages.

```
hw.c
hw.c(3) : error C2015: too many chars in constant
```

In each case the error message referred clearly to the line number in error. The reference to character constants appears because the C language uses single quotes for a different purpose (character constants).

In the third example the programmer, again possibly confused by another programming language, had missed out the semi-colon on line 3.

```
main()
{
    printf("hello, world\n")
}
```

The IBM 6150 compiler produced the following message.

```
"hw.c", line 4: syntax error
```

The SUN Sparc Station produced the following messages.

```
"hw.c", line 4: syntax error at or near symbol }  
Compilation failed
```

Turbo C produced the following messages.

```
Error C:\ISPTESTS\HW.C 4:  
Statement missing ; in function main
```

The Microsoft C version 5.1 compiler produced the following messages.

```
hw.c  
hw.c(4) : error C2143: syntax error : missing ';' before '}'
```

In all cases the error message referred to line 4 although the error was actually on line 3. This often happens when compilers detect errors in free-format languages such as C. Simply the compiler didn't realise anything was wrong until it encountered the } on line 4. The first error message is particularly unhelpful.

In the fourth example the programmer wrote *print()* rather than *printf()*.

```
main()  
{  
    print("hello, world\n");  
}
```

This, not surprisingly, produced the linker error messages shown in below. These are rather similar to the error messages shown earlier when the effects of writing MAIN() rather than main() were shown. The IBM 6150 compiler generated the following messages.

```
ld: Undefined -  
    .print  
    _print
```

The SUN Sparc Station compiler generated the following messages.

```
ld: Undefined symbol  
    _print  
Compilation failed
```

Turbo C generated the following messages.

```
Linking C:\ISPTESTS\HW.C 4:  
Linker Error: Undefined symbol _print in module HW.C
```

The Microsoft C version 5.1 compiler produced the following messages.

```
hw.c  
  
Microsoft (R) Overlay Linker Version 3.65  
Copyright (C) Microsoft Corp 1983-1988. All rights reserved.  
  
Object Modules [.OBJ]: HW.OBJ  
Run File [HW.EXE]: HW.EXE /NOI  
List File [NUL.MAP]: NUL
```

```
Libraries [.LIB]: SLIBCEC.LIB
```

```
LINK : error L2029: Unresolved externals:
```

```
_print in file(s):  
HW.OBJ(hw.c)
```

```
There was 1 error detected
```

In the final example the programmer left out the parentheses immediately after main.

```
main  
{  
    printf("hello, world\n");  
}
```

The IBM 6150 compiler produced the following messages.

```
"hw.c", line 2: syntax error  
"hw.c", line 3: illegal character: 134 (octal)  
"hw.c", line 3: cannot recover from earlier errors: goodbye!
```

The SUN Sparc Station compiler produced the following messages.

```
"hw.c", line 2: syntax error at or near symbol {  
"hw.c", line 2: unknown size  
Compilation failed
```

Turbo C produced the following messages.

```
Error C:\ISPTESTS\HW.C 2: Declaration syntax error
```

The Microsoft C version 5.1 compiler produced the following messages.

```
hw.c  
hw.c(2) : error C2054: expected '(' to follow 'main'
```

All of these messages are remarkably unhelpful and confusing except that from Microsoft C, particularly that from the IBM 6150 compiler.

You may find it interesting to try the various erroneous versions of the "hello world" program on your particular system. Do you think your compiler generates more helpful error messages?

If you are using Turbo C you will see the following message, even when compiling a correct version of the *hello world* program

```
Warning C:\ISPTESTS\HW.C 4:  
Function should return a value in function main
```

A warning message means that there is something not quite right with the program but the compiler has made assumptions that the compiler writer thought reasonable. You should never ignore warnings. The ideal is to modify the program so that there are no warnings, however that would introduce extra complications into the *hello world* program and this particular message can be ignored for the moment.

The message means that the user supplied function `main()` should return a value to the run-time support package. There is no requirement to do so and most compilers recognise `main()` as a special case in this respect. Returning a value to the run-time support package should not be confused with returning a value, sometimes known as an exit code, to the host operating system.

Questions: -

1. What are different types of programming error