Program Management

Chapter 6

Introduction

- Variety of topics in Chapter 6
- Most have to do with best practices, organizing, and managing programs

Makefiles

- So far, our programs have mostly been single files
- Easy to type in command to compile it
- What if our program consisted of a thousand files? How would we compile it?

- Could type out the command to compile all the files every time we need to compile it
  - Very tedious!
- When a program consists of many files, often use a **build tool** to compile the program
- Build tools can help automate compiling or generating the compile command to build the program
- One build system popular with C and Unix is **Make**

- **Make** uses text files, called **makefiles**, that describe how to build the program
  - Kinda like a recipe to build the program
- Makefiles are comprised of dependency / command blocks
- Each section starts with a dependency and then details the commands for creating that dependency

file : dependency
[TAB] command

- For example, if we wanted to create a program called **sqrt** that was made from two C files, **main.c** and **sqrt.c**..

```
sqrt : main.o sqrt.o
	gcc -o sqrt main.o sqrt.o
main.o : main.c
	gcc -c main.c
sqrt.o : sqrt.c
	gcc -c sqrt.c
```

- Note that the command lines **MUST** be indented with **tabs**!
- -c switch used to produce object file (.o file)
  - Remember, object files are compiled, just not linked together yet

- To actually run the makefile, we simple type the command **make** in the console
  - *Command Prompt*$ **make**
- Command searches the current directory for a file named Makefile (or makefile) and runs it
  - Usually, file name uppercase so it displays at the top of the list of files

- **main.c**, **sqrt.c**, **Makefile**
  - Program exists in two files
  - Uses Make to compile them and link them together

- One of the benefits of Make is an increase in efficiency
- Running make again will only recompile source code files that have been modified
- Building lots of files can take a lot of time, so only rebuilding the ones that have been changed can save a lot of time

Functions

- **Functions** are one of the most basic ways of making a program more **modular**
  - Allows a large problem to be broken up into multiple sub problems
- Making a program more **modular** fosters **reusability**

- Using functions can also make a program more **readable**
  - Can be difficult to follow the logic of a really long main() function
- Subsequently can make the program easier to **maintain**

Multiple Files

- Often break a large program up into multiple files
  - Another form of **modularizing**
- Helps with **reuse**, **readability**, and **maintenance**

static Keyword

- In C, **static** keyword can be applied to variables and functions
- A static function can only be called in the file in which it is defined
- A static global variable can only be referred to within the file in which it is created

- Another use of static keyword is to make a local variable who's value is persistent between function calls
  - It remembers its value between function calls

- **static_example.c**
  - Has a global static variable and a local static variable
  - Retain values between function calls

extern Keyword

- **extern** keyword can be applied to a variable
- Basically a way to access a variable that exists in another file

- **extern_example.c** and **extern_example2.c**
  - One has a global variable **x**
  - Other has extern variable **x** to refer to the x in the other file

Code Style

- Comments
  - Self-documenting code
- Block styles
  - Next of line
  - End of line
- Indentation

Enumerations

- An **enumeration** or **enumerated type** is a listing of possible values, often related
  - For example, we might make an enumeration of the days of the week, months of the year, hands in rock paper scissor, etc.
  - Values listed out are constants

- Making an enumeration is also making a new data type
- Variables of an enumerated type can only hold values that were listed in the enumeration
  - For example, a variable of the week of the day type, could only hold a week of the day constant
- Enumerations are useful for making code more readable

- Enumerations are created using the **enum** keyword
- enum is followed by the name of the enumeration, and then curly braces, listing the enumerated vales

```
enum days
{
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY
};
```

- Constants in an enumeration are really just integers, starting with value 0
  - printf("%d\n", SUNDAY);
  - ..would print 0

- **rps.c**
  - Rock, Paper, Scissors
  - enum for the different hands
  - Good practice to cast integers to enum type, even though enum constants are really just integers
  - Makes code more readable

- When referring to the enum type, have to repeat the enum keyword with the type name
- Can also use typedef to define a new type
- The typedef'd type name does not need the enum keyword

```
typedef enum
{
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY
} days_t;
```

- **rps_typedef.c**
  - Like previous example, but uses typedef for enum to make hand_t data type
  - In C, often put _t at the end of custom types to denote that it is a data type and not a variable

Macros

- **#define** can be used to create named constants
- It can also be used to create **macros**
- A macro looks like and is used like a function
- But really, it's just substituting code during the preprocessor phase

```
#define KAZAM(x)      (((x) * 3 / 4) - 2) / 5)

...

printf("%d\n", KAZAM(10));
```

printf("%d\n", **(((10) * 3 / 4) - 2) / 5)**);

- Macros are a useful replacement for simple functions
  - Especially functions used a lot
- Functions do have a slight efficiency cost, macros avoid that cost
- Overuse of macros can make confusing and error-prone code and should be avoided

Archives

- On Unix, files are often grouped together into a single file called an **archive file**
  - Kinda like a zip file on Windows
- Unix command for making an archive is **tar** and the file extensions for archive files is .tar

- Often, the **.tar files** are compressed with the **gzip** tool
- Altogether, this produces files with the file extensions **.tar.gz**

- Software for Unix and Linux is often distributed as source code
  - Many variations of operating systems, usually have to compile software from source
- The source code, including a Makefile, is usually contained in a .tar.gz file

- To create a tar file of the contents of the current directory:
  - **tar cf *file*.tar ***
- To view the contents of a tar file:
  - **tar tf *file*.tar**
- To extract a tar file:
  - **tar -xvf *file*.tar**

Packages

- Aforementioned way of software distribution is the default / most common way
  - Not the simplest way to get and install new software
- Some Linux distributions have created their own types of **packages** to make software distribution easier

- Packages are similar to archives, but the installation processes is automated / simplified by package manager software
- Red Hat's package is .rpm files
- Debian and Ubuntu use .deb / .dpkg packages