# Introduction To C

# Course Notes

# Software Engineering Technician / Technology

**Version 2014.1**

# Preface

Many hours were spent writing and making extensive changes to these course notes. This document was originally created by Ignac Kolenko and then expanded and revised by Carlo Sgro.

If you find any errors, inconsistencies, outdated information, formatting problems, or simply things that don't make sense, please e-mail with details to csgro@conestogac.on.ca.

Also, many references are made to a textbook called *Simple Program Design* by Lesley Anne Robertson (Course Technology, Cambridge, MA, 2000, ISBN: 0-619-01590-X). If you do not have much programming background in other languages, you should purchase that book.

# Simple Table of Contents

# Expanded Table of Contents

# Chapter 1. C Programming Overview

## 1.1. The History of C

C is a language developed at Bell Labs in the early 1970's. Its early roots were closely tied to the creation of the UNIX operating system for mini and mainframe computers, and because of this history, it has historically been the standard language for the development of most new operating systems for computers.

C was created to help solve the problem of designing and writing an efficient **operating system** for a computer. The operating system of a computer is the software, which allows the outside world to control the hardware within the computer system, much like Windows allows a user to manipulate resources such as disks on a PC.

Operating systems are naturally required to be fast, efficient, have complete control over the hardware resources of the computer, and most of all, be easy to upgrade and maintain even years after they were written. C is a language that supports those ideals.

Over the years, standards committees have been created to help arrive at a version of the C programming language that is common to all computing platforms throughout the world. This version is commonly referred to as the ANSI Standard C language (ANSI is an international standards committee), and you will find most C compilers are ANSI-compatible (there are more modern standards that are not as widely used).

## 1.2. Why C?

C is a language which allows the author of a program to communicate with the machine. The difference lies in how this language causes a computer to execute a desired function. C is a compiled language, whereas many other languages (e.g. JavaScript) are interpreted.

"Interpreted" means the computer is running a program which will translate the program instructions into something the computer understands. The interpreter program will do this operation each and every time you run your JavaScript program. Consider a program to generate a conversion table to convert degrees Fahrenheit to degrees Celsius in JavaScript. If you ran this program one hundred times a day, every instruction in the JavaScript program to handle this conversion will be translated each and every time the JavaScript interpreter sees the instruction, even though it may have translated it just seconds earlier. This act of constantly interpreting instructions cause interpreted languages to be relatively slow compared to the full speed of today's computers.

C, on the other hand, is a compiled language. A compiled language means that you will have to run a program, called the **compiler**, to translate your C language program, which is normally called **source code**, into something called an **object** file. The object file contains machine-readable code, but not in its final form – in other words, it is not fully executable yet, as it is missing vital information to allow your code to link to existing functionality. Another program, called the **linker**, will combine groups of object files into an **executable** file, which is then a program you can execute on your computer, much like

other commands in your operating system. The translation has been done exactly once for you, and if you want to run your C program a thousand times each day, you only have to run the executable version of your C program, not the compiler! It should be noted that when programmers wish to reuse large numbers of object modules from application to application, often a program called a **librarian** will be used to manage libraries of object modules for code reuse. When testing out your executable files, another program, called a **debugger** will be used to help test out your application, to see where problems may exist, and to help trace through areas in your software that require special attention. This then allows a programmer to return to the editor, make changes, recompile and hence, repeat the cycle once again. Quite often, all of these tools are combined into an **Integrated Development Environment (IDE)**. As an example, Visual Studio contains an IDE for Visual C++. You can do all of your editing, then compile, then link, and finally execute your program, all within the IDE.

A compiled language such as C creates programs that are **fast**. The compiler will translate the C program into instructions which are handled **directly** by your computer's CPU, and once the translation has been done for you, the computer can execute these instructions at full speed each and every time you run your compiled program.

As can be seen from our above discussion, C is a language that can be used to generate fast, efficient programs which allow you to exploit the full speed of your computer, something an interpreted language like JavaScript will never allow you to do. This is the main reason why operating systems such as UNIX are written in a language like C.

A slight drawback of C, which is due to its compiled nature, is that you do not know what your program will produce as its results until after you run the executable file produced by the compiler. The compiler is only capable of translating the program into an executable file. If there are any grammatical mistakes, commonly referred to as **syntax errors**, in your source, they are found and reported, and the compilation is aborted to give you a chance to fix the errors. If no such grammatical mistakes exist, an object file is created. The linker will in turn generate an executable file on your computer's disk. At this point, you are finally able to see the fruits of your labour. You may execute your program, and if something is still incorrect (for example, a logic problem), you must fix it, re-compile, re-link and re-execute your software once again. If more mistakes exist, the whole process is repeated. The debugging cycle for a C program can become rather lengthy!

## *1.3. No, Really … Why C and not C++ or Java or C#?*

C is recognized as an excellent language for learning the fundamentals of programming and still giving you the skills to pick up other programming languages very easily. It doesn’t have a lot of the overhead of object-oriented languages and doesn’t have some of the more advanced features of other languages like automatic garbage collection (you’ll learn about this when you learn about pointers). If your first language has something like automatic garbage collection, you will have problems getting used to situations where you **don’t** have that feature (e.g. in an embedded system where you’re working with C in

a very small code and data space). Since C is the basis (directly or indirectly) for C++, Java, and C#, if you can learn C, you can learn anything based on it very easily.

## 1.4. Compilers

Conestoga College has made arrangements with Microsoft to provide their Visual Studio development systems for free for you.

MSDN-AA allows you to access Visual Studio (which contains Visual C++), as well as many other Microsoft products (such as Windows 8, Windows 7, Windows XP, OneNote, Visio, and Virtual PC). In some cases, these products come in .ISO format, which requires that you burn it onto a blank CD or DVD (or mount the .ISO using a third-party utility). In other cases, a proprietary downloader will be used to download the software.

Download of Visual Studio Professional is rather large (over 2 GB) and installation requires over 3 GB of hard disk space. Typically, newer versions of Visual Studio (e.g. 2012, 2013) end up being larger. If the space requirements are too much for your computer, Visual C++ Express is smaller and still acceptable. However, there is no guarantee that Microsoft will continue to make Visual C++ Express available for free download.

We will use Visual C++ (through Visual Studio) in class. There are also other compilers available for other platforms. If you use other compilers, it is required that anything that you hand in must be compilable under Visual C++ and runnable under Windows 7 in console mode.

## 1.5. Design Cycle

The following table will highlight the design cycle that a typical C program will follow:

| Program | Input | Output | Function |
|---|---|---|---|
| Editor | keyboard input | .c (source), .h (header) | create and maintain source code |
| Compiler | .c | .obj | take source code and translate it into an object file |
| Linker | .obj, .lib | .exe | combine multiple object files into an executable file |
| Debugger | .exe | n/a | allows debugging of your program |

The Visual C++ IDE incorporates many elements of the design cycle mentioned above. First and foremost, it has a **project manager** that allows you to set up the type of program that you want to write. Secondly, it has an **editor** so that you can create the C source files that you will work with. It also has a compiler and a linker to create the program. At the bottom of the IDE, there is a **Messages** window that contains status and error messages associated with the compile and link. Lastly, it contains the ability to **run** and **debug** the program that you have created. The details on how to create a program from the setting up of the project to execution is found in Appendix A. Debugging is discussed later.

## 1.6. Common Beginner Errors

- Make sure that you set up a project appropriately with the correct defaults.
- If you are having problems with your project, make sure that you follow the instructions found in the appendices of this document.

## 1.7. Summary

This chapter has highlighted the basic underlying concepts regarding the C programming language. C is an ideal language to create and maintain computer operating systems, to manipulate low level hardware within a computer system, and because of the data structure features of the C language, it lends itself quite easily in database manipulations.

C is a well-rounded language in that it offers much to just about any application that can be thought of. There may be other languages which offer better ways of achieving the

same goals, but C at least allows you to attack any programming problem with few overriding concerns because of its flexibilities.

We've also looked at how to create a C program, using an IDE through separate elements of the design cycle.

# Chapter 2. Source Code Structure

## 2.1. A Typical C Program

As was alluded to in the first chapter, a C program begins its life as a **source code** file. The source code is a text file, which contains C statements, grouped in such a manner as to create a program that can be compiled by the compiler. The compiler must take the text from the source code file and translate these instructions into code that the microprocessor will execute. This will allow you to run the software you have created.

There is a fairly standard appearance to typical C code. Just about all software written in C will follow this order:

- Header comments
- Include files
- Constant definitions
- Type definitions
- Function prototypes
- Global variables if permitted
- Function definitions

To help us understand the above concepts, a sample C program will now be introduced.

```
/*
 * Filename:    file.c
 * Project:     assignment1
 * By:          Joe Schmoe
 * Date:        August 13, 2000 (yes, this is old)
 * Description: This program is a simple demonstration
 *              of a C program.
 */

#include <stdio.h>

int main (void)
{
    printf ("wow. a C program\n");
    return 0;
}    /* end main() */
```

## 2.2. Program Structure

### Header Comments

You should notice that prior to the actual statement which includes **stdio.h**, there are a number of lines of code which begin with a slash-asterisk pair (/*****) and end with an

asterisk-slash pair (***/**). The two pairs of characters are called the **comment delimiters**, and are used by the C compiler to designate when you are adding comments inside of your code. The comments are meant to help you document what you are trying to do with your code, and are invaluable in situations where someone else will be required to maintain the software you have written. It's always easier to read a programmer's comments rather than try to always figure out what the programmer did by leafing through pages of code! Note that modern compiler packages are dual C and C++ compilers, and will thus accept the C++ comment style: // (a pair of slashes side by side).

Every C file that you write should have a comment at the top of the file (called the **header comment**). It is not required in order for the program to work but it is required so that you and others can tell a bit about the contents of the file. Typical contents include:

- filename
- project name (if different from the base of the filename)
- programmer's name (the first person to work on the file, typically)
- date of the file's first useful version
- description of the contents (this is important)

**You will see many boxes like this in the course notes. They indicate Course Requirements, Useful Tips, and other important things you should know.**

**In the case that there is a conflict between a Course Requirement found in this document and some other document provided to you, the other document will have precedence unless your instructor tells you otherwise.**

**Course Requirement: Header Comments**

You must have a header comment in each source code file that you create. It must contain the information mentioned above. The format should be similar to that shown in the example above.

**Course Requirement: Header Comment Description**

The description contained within the header comment should accurately describe the **intent** of the contents of the file. This is most important in multi-file projects that contain many, many .c and .h files. Thus, a description like “This is assignment #2.” is bad while a description like “This program takes Fahrenheit temperatures from the user and displays them as Kelvin and Celsius temperatures. The program loops until the user enters an invalid temperature.” is good.

Besides the header comment, you will typically have comments associated with the code itself. As a rule of thumb, always try to make your comments as plain and judicious as possible. Don't write a Shakespearean soliloquy when a simple statement would suffice.

Also, try not to comment every single line of code, rather only those lines of code that perform a function that are not immediately obvious. C coding is, by its very nature, generally self documenting.

The examples in these course notes will not have many comments, as the text of the paragraphs surrounding the examples should serve the purpose of the comments.

## Include Files

Just about every C program has a section where special files called **header** or **include** files are declared to be compiled along with the source code stored in the C file itself. These header files, many of which are supplied by the compiler manufacturer, are necessary because they contain important constants and declarations to allow your C program to properly access the **standard library** of functions. In Program 1, you will notice we have used the statement:

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

to tell the C compiler to include a file called **stdio.h** (pronounced either "standard-i-o-dot-h" or "stid-i-o-dot-h") when compiling Program 1. This header file contains declarations for the standard input and output family of capabilities, which allow the C programmer to obtain keyboard input, and send output to the computer console (monitor). These input and output capabilities (usually called **I/O functions**) have been designed by C compiler manufacturers to be common to all implementations of the C language, and any software written to use this header file are (more-or-less) **portable** to any other computing platform.

Whenever you add statements to tell the C compiler to include header files within your own software, be aware that you are in reality asking the C compiler to temporarily pretend that the statements within the header file actually belong to your source code. Don't add header files to your source code unless you absolutely need the statements that they contain. You will find that the standard header files for most C compilers are designed in such a way that the statements they include are grouped by purpose. For instance, if you want to manipulate mathematical equations, you can include **math.h** or, if you want to work with character based strings, include **string.h** with your source code.

**Useful Tip: Header Comments**

Header comments are not comments about the header (or include) files. This is especially important to know for exams!

## Definitions, Globals, and Prototypes

After header file inclusion statements we have sections where we have various definitions, global variables and function prototype declarations. As our first sample program does not require these features, we will leave this section for the time being since it is not a concept that is crucial to our understanding of C program format at this moment. It suffices to know at this point that a global variable is a piece of **memory** that can be used to store information your program will manipulate, and that a prototype informs the compiler of a function before it is defined in your source code.

## Functions

What comes next are the **function definitions**. This is where the actual work your program will accomplish resides. A function in C can be considered to be **a block of code which accomplishes a specific task**. You will find that C functions are as diverse as calculating the square root of a floating point number, or printing characters to the screen or reading a file in from a disk. Each of the above actions are a specific task, and as such, are easily handled by the creation of simple functions which are easy to use and understand.

# *2.3. Functions*

## The main() function

The only function in Program 1 is the function called **main()**. (Note: When discussing functions, C programmers usually denote the name of a function using the above notation: the name, followed by a pair of round brackets. This easily distinguishes a function name from a variable name when discussing source code.) This function is mandatory in the sorts of C programs that you will be writing for this course. You cannot fully compile and link a C program into an executable file without having a function called **main()** somewhere in the source code you have just compiled. This is due to the manner in which a C program executes once the operating system has transferred control to your software - the operating system will in essence jump to the entry point inside your application called main().

A C program always begins to execute on the first line of code within the **main()** function. This line of code must exist **after** the opening brace (sometimes called a "squiggly bracket" in informal C jargon) and **before** the closing brace. In Program 1, the first line of code inside the **main()** function is the line which reads:

```
printf ("wow. a C program\n");
```

## Standard Library Functions

This line of code is actually telling the computer to execute a task which is known as a **standard library function** for virtually all C compilers: **printf()**. This function has the job of displaying characters on the computer's monitor. The characters it will display are placed inside of the pair of double quote (") characters, therefore the output we will see on the screen is:

```
wow. a C program
```

The use of the double quotes and characters between them are an example of a **construct** commonly found in C programs: a **string literal**. The entire string as seen above constitutes a **parameter** for the function **printf()**. A parameter can be thought of as information that flows into a particular function. The function itself will take the input and process it and, at times, return some information back to the line of code which called the function.

**Useful Tip: Function Concept**

A function can be thought of as a black box with information flowing into the box, and information flowing out of the box. The black box will process the incoming information and transform it to the output. Generally, from a programming point of view, one does not need to know exactly what happens "in the box", just understand how your input is transformed to the output.

**Useful Tip: Arguments and Parameters**

Parameters are typically called arguments when they are passed in the calling function. It is not unusual, though, to refer to arguments as parameters as well.

Once **printf()** has printed to the screen the string parameter that it has been supplied, the next line reached is the **return 0** line. This returns control from the function. This will be discussed later when we discuss functions in more detail.

Finally, we find that we come across the closing brace. This informs the C compiler that this function is complete. Because we have just terminated the **main()** function, we are actually terminating our C program as well. When **main()** completes, we are returning control to the operating system or the compiler's development environment.

Although Program 1 is a very simple program, it is a perfectly valid example of how a C program is constructed. All C programs begin to execute instructions contained in the **main()** function. It is up to the programmer to decide what to place inside of the **main()** function, thus **main()** is never pre-defined anywhere in any of the compiler's libraries. Execution of the C program will stop as soon as there are no more instructions to process within the **main()** function. A programmer may wish to put as little as 1 or 2 instructions (as we have above) or millions (or more) statements within the braces of the **main()** function. The latter example usually would prove to be far too unwieldy for most programmers to handle, and as you will learn in later chapters, you are free to create your own programmer-defined functions apart from **main()**. This feature allows streamlined, modular C code to be produced with the minimum of fuss.

**Course Requirement: main() Return Value**

The **main()** function **must** return an **int** value. If you don't care what the value is (and you typically don't), you can feel free to use **return 0** before the closing brace in **main().**

**Don’t return void! Don't leave out the return value!**

## Function Declaration Syntax and Style

The general format for a function can be described using the following declaration notation:

```
returnDataType nameOfFunction (parameterList)
{
datatype firstLocalVariable;
. . .
datatype lastLocalVariable;
```

```
    firstLineOfCodeForFunction;
    . . .
    lastLineOfCodeForFunction;
}    /* end function */
```

Notice the **style** of the sample code above: the return data type (to be discussed below), function name, and parameter list are on the first line, against the margin. The opening brace is on the second line against the margin. This layout is not mandatory but it is the style that will be used in these course notes.

The body of the function (local variable declarations and code) is **indented** one tab stop. The closing brace is back at the margin. This is a very common C programming style. There are different styles that can be used (to be discussed later). The C compiler doesn't care what style is used, but you should adopt a particular style, and stick with it.

**Whitespace** (blank lines, spaces, etc.) is highly encouraged when writing programs. It breaks up a page of code into visually meaningful sections. Notice that in the sample notation above, the local variable declarations (discussed in the next chapter) have a blank line between them and the code of the function. This separates code from any variable declarations.

## Indentation and Bracing Style

For the most part, the C compiler does not care what indentation and bracing styles you use. It's the people who you work with who have to look at and change your code that care what your style looks like.

Why is style important? An absolutely incredible book called Code Complete: A Practical Handbook of Software Construction by Steve McConnell (Redmond, Washington: Microsoft Press, 2004) uses what it calls The Fundamental Theorem of Formatting: "**Good visual layout shows the logical structure of a program.**" To be more specific, a good layout "should **accurately** and **consistently** represent the logical structure of the code, improve **readability** of the code, and withstand **modifications.**"

The primary feature that should be present in indentation and bracing style is **consistency** within the code written in your functions. If you choose a particular style, then keep to that style in that file. You will also find it best to be consistent across many files, so that anyone looking at your code will have an idea of what to expect.

Here's some instructions on what to do:

- use white space to **group related items** together, using blank lines, spaces within lines, and indentation
- indent every time you make a **decision** in your code (e.g. if, while, for, do-while, switch) since you may or may not be executing the code in the corresponding block
- indent the bodies of functions so that they are not level with the left margin
- indenting one TAB for each code level is the best way to ensure consistent indentation; you can usually adjust the number of spaces that each TAB stop represents in the customization section of the editor you are using

### Three bracing styles

#### Style 1: Braces level with code block

```
int main()
{
   printf ("wow. a C program\n");
   return 0;
}
```

#### Style 2: Braces level with control statement

```
int main()
{
   printf ("wow. a C program\n");
   return 0;
}
```

#### Style 3: Braces attached to control statement (K&R Style)

```
int main() {
   printf ("wow. a C program\n");
   return 0;
}
```

### Flexibility in Indentation and Bracing

In certain cases, it does not make sense to require a style that might actually detract from the understandability of the code being written. One of the primary places where variation happens is variable declaration and initialization.

- If you're using style 2 or style 3, it's typically considered OK to indent a variable declaration **either** level with the brace that surrounds the code block **or** with the code block itself, like this:

```
int doIt(void)
{
int i = 0;            /* level with brace */
    printf("i is %d\n", i);
    return i;
}
```

or this

```
int doIt(void)
{
    int i = 0; /* level with block */
    printf("i is %d\n", i);
    return i;
}
```

Again, though, you must be consistent.

- If you have to use braces to initialize a variable (e.g. with a struct or array), it's most desirable to use a bracing and indentation style that best helps with the

understanding of the data. Quite often, that means simply having the data on the same line as the variable declaration, like this:

```
int lotsOfStuff[5] = { 4, 54, 2, 3, 1 };
```

A typical question at this point is "why show us stuff like arrays that we haven't even seen yet?" The answer is that it is important to not develop bad style habits so you're hit with the right way to do it right up front.

**Course Requirement: Style Consistency**

You must use correct and consistent style in this course!

## Data Type

The term **data type** may be unfamiliar to most novice programmers. A data type is usually defined as the type, or format, a particular piece of information may have within a program. For instance, the number **pi** (3.1415926535...) is said to be a **floating point** data object since it is a real number (it has decimal digits following the whole number part and decimal point). The letter **A** is usually said to be a **character** data object because the letter A is definitely a character of the alphabet. The number **37** is an **integer** data object since it is a number without a decimal point (integers contain no fractional portions).

For our **main()** function inside Program 1, you will notice that we have declared the return data type and parameter list of **main()** to be **int**. This indicates that an integer value will be returned to the caller of **main()**, which is usually the operating system. In the next chapter, we will have a close look at the different data types that are standard in C.

## Function Name

After defining the return data type, a function requires the **name** to be defined. The name of a function can consist of any combination of alphanumeric characters as well as the underscore character, although a name cannot begin with a number and at least one alphabetical character must exist in the name. Valid function names include:

```
test1()
_123Execute()
a()
```

A word of warning: you must always define **unique** names for your functions (and variables). Two functions with the same name in the same source code file will lead to compilation errors, forcing you to rename one of the offending functions. Also, different compilers have different restrictions on the maximum lengths of names. This is typically not a problem, since the lengths are usually large enough.

Also, it should be noted that C is **case-sensitive**. That is, uppercase and lowercase letters are deemed to be different. Thus, the following function names are all different:

```
doIt()
doit()
DOIT()
DoIt()
```

In this case, however, it would be a very bad decision to create more than one of the above functions in a program. A great deal of communication in the workplace is verbal: you don't want any **ambiguity** if someone tells you "just take a look at the do it function".

For the same reason, you should always be consistent in how you use case when you name functions and variables. If you tell a co-worker "just take a look at the do it function" and they know your style of programming, they should be able to immediately know what the spelling of that function name is.

Start all of your function names with a lowercase letter. This will help you differentiate functions from C++ methods when you program in C++. The language will let you use either camel case (e.g. doIt()) or underscores (do_it()) to separate words in function and variable names. That way, your names will be easier to read and understand. The use of underscores is rapidly becoming obsolete, however.

**Course Requirement: Distinguishing Between Words In Identifiers**

Use camel case to distinguish between words in names, unless otherwise explicitly stated.

**Course Requirement: Naming**

Start your names with a lowercase letter, unless otherwise explicitly stated.

Although the standard syntax for a function suggests that we must define the names and data types of the parameters for the function, you will find that not all functions require input parameters. These functions will therefore take a **void** parameter list, as our example in Program 1. **void** is used to indicate that no information will be flowing to the function. If you are designing a database program, and a common help screen is required from a number of different places within the program, you may want to put the help code inside of a help function. Such a function does not need input: its only job is to display help information on the screen, therefore the parameter list will also be **void**. It probably will not return a value to the caller of the help function, therefore its return type would be **void** as well (indicating that no information will be flowing back from the function to the caller). We will see more examples of data types and variable names in the next chapter.

One final note: many statements within any function require the programmer to append a **semicolon** to the end of the statement. This is the method the C compiler uses to delineate when one statement is complete and another begins. This is an important concept to understand and a useful concept since C does not require a complete statement to fill one line. In fact, a valid example of a C program statement includes:

```
printf ("testing %d %d %d",variable1, variable2,
     variable3);
```

Even though the statement above takes two lines of code, it is still only one single statement to the C compiler, since the semicolon is at the end of the second line of code.

**Useful Tip: Semicolons**

Figuring out when semicolons are required in a C program can seem like a black art. Here's some general guidelines, some of which use terms that will be described later in the notes:

Use a semicolon:

- variable or type declarations
- function prototypes
- function calls that are alone on a line
- normal executable code statements that do not have a code block after them

Don't use a semicolon:

- comments
- "#" statements
- function declarations
- "{" or "}"
- statements that usually have a code block after them (e.g. if, while, for, switch)
- statements within function calls or conditions

**Useful Tip: Finding Compile Errors**

If the C compiler generates an error which does not appear to make sense at the line number where the error is reported at, generally it is advisable to look at statements just *prior* to the reported line number. Often, a missing semicolon is the culprit prior to where the compiler got confused.

A C compiler will attempt to recover from an error and make assumptions about what you had intended. Often, the assumptions it makes lead it to other compiling errors. So, do not be alarmed if a single missing semicolon causes hundreds of errors to appear - it really isn't that serious!

## 2.4. Another Sample Program

Here's another sample program which will be discussed in class:

```
#include <stdio.h>

int main(void)
{
int number = 9;

    /* print the number upon startup */
    printf("Numbers are coming... ");
    printf("The number is %d\n", number);

    /* change the value of the number and
```

```
     * print it */
    number = 8;
    printf("The number is now %d, not %d\n",
        number, 5);

    return 0;
}
```

**Useful Tip: Code Samples in this Document**

You will find that your program runs too quickly and then disappears. If this happens, you can choose to execute the program in a different way. If you choose *Start Without Debugging* (which is, oddly enough, found in the Debug menu), your program will pause at the end. This will, as the name suggests, not allow debugging, though.

## Changing The Sample Program

Once you have compiled and run the program above, do the following to the program:

**Make a minor change**

Change int number to int fred.

Execute the program.

Note the error.

Why did the error occur?

Fix the error.

Execute again.

**Syntax change**

Put a semicolon after the *int main(void).*

Execute.

Note the errors.

Note that the errors aren't all that clear in terms of helping you solve the problem.

Undo the change.

Execute.

**Comment change**

Get rid of the */ at the end of the first comment. Note the change of colour.

Execute.

Note the change of result.

Undo the change.

Execute.

**String change**

Delete the closing " in the first string.

Execute.

Note the error.

Undo the change.

Execute.

**Bracing change**

Delete the closing }.

Execute.

Note the error.

Undo the change.

Execute.

**Review errors**

Note the error wordings and how they related (or didn't relate) to the causes.

Note the error locations and how they related to the actual locations of the causes.

**Useful Tip: Error List**

Start compiling a list of common errors. Share the list with your friends.

Yes, I'm serious.

## 2.5. Common Beginner Errors

- Make sure that you remember your braces and semicolons.
- Close all comments using */ or use the // style of comment.
- Put all code that you want to execute within a function (for now, within main()).

## 2.6. Summary

In this chapter we have learned a number of useful concepts, of which the idea of a **function** is the most important. All work that a C program will accomplish is through the use of one or more functions, each performing a certain task. Every C program that we'll be doing in this course must have at least one function called **main()** somewhere in the source code. A C compiler cannot produce a complete executable file without **main()** existing in the source code.

Other useful concepts include the use of **header** files in our source code. A header file consists of definitions of important constants and functions which allow us to take advantage of the features in the standard library of functions.

A function will begin execution after the opening brace of the function, and will cease execution prior to the closing brace of the function. The **main()** function contains the first and last lines of **all** C programs, which makes this function the most important function of all.

Style is very important, as source code that is hard to follow is bad source code.

We also looked at some specific ways of working with C, compiling and making corrections to fix errors.

# Chapter 3. Variables and Data Types

In the previous chapter, the concept of variable declarations, although standard parts of any C program, was left until this time. This was due to the fact that a fundamental understanding of what a variable is in C, as well as the type of memory storage allocated to each data type must come first before learning how to declare a variable. Firm knowledge of data types and storage requirements will allow a competent C programmer to write efficient code based upon the data requirements of the program (data driven software) rather than basing code on what the language forces you to use (code driven software).

In C, there are a number of fundamental data types already defined for the programmer to use. These data types are based upon the standard building blocks the microprocessors and co-processors of today use. The data types are thus oriented around what a processor can access efficiently, and as such, they may appear to be strange to a budding C programmer.

A variable is simply a piece of memory that has been allocated (obtained) from the overall memory of the computer, for the purpose of storing values that a program may wish to record. Variables are useful in storing values based upon computations a program generates, and thus, some knowledge of the various types of data objects is necessary before a C programmer can define any variables.

## *3.1. Data Objects in your CPU*

Virtually every computer can access the memory inside of the **central processing unit** (CPU) in the form of **bytes**. A byte is normally the smallest unit of memory the processor can accesses at one time, and it consists of 8 **bits**, each bit representing the number 0 or 1. This is the same type of information a light bulb stores - it's either on or off, zero or one. With eight bits of information within each byte, it can easily be shown that there are 256 possible combinations of bits within a byte, thus the byte can store a number between 0 and 255 inclusive (a range of 256 values).

The next step up from a byte is the **word**. A word is a pair of bytes side by side in memory on PCs, which gives us 16 bits of information at one time. With 16 bits, a word can have 65536 possible combinations of values, thus the range of values that can be represented within a word are zero to 65535 inclusive.

Next, computers work with **long words**. A long word consists of a group of four bytes side by side in memory accessible at one time, a total of 32 bits of information to manipulate at any given time. With 32 bits, values from 0 to well over 4 billion can be represented. Some other microprocessors also use the **quad word**, with 64 bits of information.

With the above three different data objects, an important fact must be recognized: bytes, words and long words are all **integer** data objects. The term integer data object refers to the fact that only whole numbers can be represented by the value stored in such a data object, with no fractional part. Thus, 3.5 cannot be stored in any combination of bits since

the fractional portion, .5, cannot be represented. Another data object (to be described below) allows a C programmer to avoid this pitfall if decimal fractions are desired.

## 3.2. Signed vs. Unsigned

Another important fact about the integer data objects is that if the highest bit (or **MSB** - most significant bit) of the particular data object is set to a value of one (on), a CPU has the ability to work with that number as either a **signed** (both positive and negative numbers) or **unsigned** (only positive numbers). In either case, the total number of different values does not change, only the interpretation of the values that can be represented. For instance, for a signed word, values between -32768 through 32767 can be represented. This still gives us 65536 different possible values, only the meaning of that high bit has changed. CPUs have the ability to work with both positive and negative integer values using the standard byte, word and long word data objects.

It should be apparent to a budding programmer that, depending on the nature of the software that must be created, a wealth of data objects can be manipulated, each with its own natural range of values. Imagine a program that is required to maintain votes for a national referendum in Canada. By studying the data requirements here, a programmer should realize that it could be possible that approximately 15 million adults can vote in total, thus to be able to store the count of all votes for a yes vote or a no vote, a long word will do the trick. There could be 15 million yes votes or 15 million no votes, or a combination of the two. By trying to save memory and using a word or byte, a programmer will be inviting possible errors if more than 256 or 65536 people vote in the referendum. A data object that is too small to hold the number will **wrap around** or be **truncated** after its natural limit is reached. On the same token, to keep track of the speed of a standard passenger automobile, a program can use a byte since there are virtually no street-legal cars that travel more than 256 kilometres per hour!

## 3.3. Floating Point

If you can recall high school mathematics, a **real** number is a value that has both a whole number portion and a possible fractional portion as well. The number (3.14159 or pi) is a floating point, or real number, since it has a whole number portion (3) and a decimal fraction portion (0.14159). We can't express most real numbers using integer values because of that fractional portion. We can, however, deal with them as **floating point** values.

These are special data objects for most microprocessors since they are not naturally equipped to deal with such values. CPUs are designed to work with integer-based values with high speed, and only deal with floating point values using special routines in whatever software is running.

A floating point data object has the capability of storing values that far exceed the normal range of values for a long word. In fact, the typical floating point number today is represented as a sequence of 8 bytes, with the 8 bytes having special meaning to the floating point co-processor. With such a large number of bytes used to store floating point values, a programmer can count on the range of values to be quite large. In fact, the

usual range for a floating point value is approximately $1.0 * 10^{308}$ to $-1.0 * 10^{308}$. There is a discontinuity near 0, as well, as the smallest positive value that can be expressed is usually $1.0 * 10^{-308}$ (which is 1 divided by $10^{308}$). Naturally, the smallest negative value that can be expressed is $-1.0 * 10^{-308}$. Each compiler can have different values for the endpoints of these ranges.

You should also note that a floating point number is **always** a signed value: there is no unsigned counterpart as in the integer data objects.

## 3.4. Compiler's View of Data Objects

The C language has been designed to take advantage of a computer's internal architecture so that efficient use of memory can be obtained. Thus, the standard data types available to a C programmer, which are predefined by C, are based upon the data objects described above.

The byte data object is called **char** in C. A **char** is by default a signed data type, thus a variable of this data type can store values from -128 through 127. A C programmer can also declare a variable to be the unsigned version of a byte by using the data type **unsigned char**. In fact, placing the keyword **unsigned** in front of any of the integer data types (the rest to be described below) can produce the unsigned or positive only versions of the standard integer data objects. Both **char** and **unsigned char** take one byte of storage.

The word data object is defined as **short**. It too is signed, allowing a variable of this type to hold values from -32768 to 32767. Again, the unsigned version is simply **unsigned short**. Both **short** and **unsigned short** take 2 bytes of memory.

The long word is defined as **long**, and is a signed data type. Its range is from less than -2 billion to over 2 billion. The unsigned equivalent is **unsigned long**. Both **long** and **unsigned long** take 4 bytes of storage.

In C, the data type called **int** is defined to be the *natural word size* of your CPU. In other words, the most natural processing data type that your CPU is geared to work with is assigned to be an int data type in C. An **unsigned int** is therefore the positive-only version of this data type. For some CPUs today, an int and an unsigned int are 32-bit values. For other modern CPUs, they are 64-bit values. The current version of Visual Studio uses **32-bit** ints and unsigned ints, regardless of what the CPU is.

The floating point data object has two different representations in C. The first is called **float**, which is a **single precision** floating point number representation. The term "single precision" is used to denote how many significant digits the data object can hold, with single precision typically being able to hold about 7 or 8 digits of precision, much like a typical calculator display. This is more than enough for a typical household budget, but nowhere near enough to calculate accurately where the next satellite will be placed into orbit.

To alleviate this problem, C has another version of the floating point number called a **double**, which, as you may have already guessed, is capable of storing twice as many significant digits as the **float**, normally 14 to 16 digits of precision.

Although both **float** and **double** can hold floating point values, it is advisable not to use the **float** data type as most math co-processors use the **double** data type automatically. Extra code must be created by a compiler to convert a **float** to a **double** prior to invoking an instruction on the floating point co-processor. The only advantage a **float** has over its **double** counterpart is that the former takes only four bytes of memory, while the latter uses eight. Therefore, in cases where memory limitations dictate data type choices, a **float** may be useful. For most applications, use the **double** as the standard floating point data type. Again, different compilers will have different values for the sizes of the **float** and **double** datatypes.

The following chart may help you to decide upon what data type is applicable in your programs.

| **Data Type** | **Bytes (typical)** | **Range** | **Values** |
|---|---|---|---|
| char | 1 | -128 to 127 | 256 |
| unsigned char | 1 | 0 to 255 | 256 |
| short | 2 | -32768 to 32767 | 65536 |
| unsigned short | 2 | 0 to 65535 | 65536 |
| long (or int) | 4 | -2147483648 to 2147483647 | 4294967296 |
| unsigned long | 4 | 0 to 4294967295 | 4294967296 |
| float | 4 | $-3.4 * 10^{38}$ to $3.4 * 10^{38}$ (with a discontinuity close to 0) | infinite |
| double | 8 | $-1.7 * 10^{308}$ to $1.7 * 10^{308}$ (with a discontinuity close to 0) | infinite |

## *3.5. Notation*

To actually create a variable in C (called "declaring the variable"), the code must follow the following syntax:

```
dataType variableName;
```

The first portion of this declaration informs the C compiler of the data type the variable is composed of. The C compiler must know this information before it allows you to use any variables in your program since the C compiler must generate code to allocate memory resources for your software.

The second portion of the declaration is the name itself. The variable name follows the same naming rules as does the name of a function. The following variable declarations are valid in C:

```
int x;
double _fpSeg;
char keystroke;
long a, b, c;   /* see next paragraph for a commentary on this */
int flag = 0;
```

Notice the second last declaration. Here we have instructed the C compiler to reserve three long words of memory, named a**,** b, and c. When you have more than one variable with the same data type, the compiler will allow you to put them on the same line of source code, separated with commas. This is often considered bad form and some other languages do not allow this, though, so it is not allowed under our SET Coding Standards.

Notice also that all variable name declarations, when complete, are terminated with a semicolon, which **must** be present for the C compiler to accept the declaration as valid.

## *3.6. Initialized Variables*

In the above examples, the final variable, flag, was followed by an equal sign and the number zero. This is an example of **initializing** your variables at the time of its declaration. This is a common (often required) programming practice, and is an indicator that the programmer understands the default values of variables in their program. Many compilers do not guarantee what the default values of variables will be, so it is up to the programmer to initialize variables with their proper values as often as possible.

**Course Requirement: Initialization of Variables**

Always initialize your variables **when you declare them**, unless notified otherwise. Failing to do so will cause you to lose marks and get quite upset about it.

## *3.7. Global Variables*

In the previous chapter, it was stated that variables may be declared after the header file inclusion statements and before any function declarations. Variables defined at this point of a C program are termed **global** variables because the inventors of C deemed that variables defined outside the **scope**, or sphere of execution of a function (outside the open and close braces of a function), are allowed to be accessible within **any** function of a program, hence the term global.

Global variables are very easy to use but are also extremely dangerous. Since they can be changed anywhere within a program, they quite often make debugging logic problems horrendous. Their use should be heavily restricted. It is a good idea to have global variable use reviewed by other members of the programming team.

**Course Requirement: Global Variables Banned**

**Do not use** global variables unless explicitly permitted by the instructor. Note that this does not restrict the use of global **constants** (variables whose declarations are preceded by the keyword const; this is to be discussed later).

There is a LARGE mark penalty that is associated with using global variables when not allowed to.

## *3.8. Local Variables*

It is often the case that a programmer may want to make a variable accessible only during the execution of a particular function. This type of variable is called a **local** variable, since its accessibility (called **scope**) and lifetime is limited to the execution of a single function. Local variables are usually declared after the opening brace of a function and before the first line of code and only that one function can access those variables.

It should be noted that a local variable can have the **same** name as a global variable, but the local variable will take precedence over the global inside of that function. Having local variables that have the same names as global variables is A Bad Idea. Having local variables in several functions that have the same name is a common thing and is A Good Idea.

Global variables can be useful for storing information of a overall nature in terms of software execution. For instance, a word processor may want to globally store the state of the Insert key on the keyboard since all portions of the word processing software must know if it is acceptable to insert new characters instead of overwriting old characters. Another example would be storing the default drive and path information for the loading and saving of file oriented information from a program. However, in most normal cases, global variables can be **eliminated** through proper design and choice of programming language. You will see the exceptions to this in some second year courses.

Local variables are useful for storing intermediate results of computations during the execution of a function. If a function was written to calculate the average of a series of numbers, a local variable may be created to hold the running total of the series prior to averaging.

To help clarify these topics, two short programs will be introduced. Both programs declare the same variables, although Program 2 declares them globally, and Program 3 declares them locally. The aim of both programs is to calculate the average of two numbers.

```
/*
 * Program 2 - PROG2.C
 * NOTE to student: to save space in this document, full
 * header comments will not be used.  You must still
 * provide them in assignments, though!
 */
```

```
#include <stdio.h>

/* global variables */
int a = 0;
int b = 0;
int c = 0;

int main (void)
{
    a = 5;
    b = 3;
    c = (a + b) / 2;

    printf ("a: %d b: %d c: %d\n", a, b, c);

    return 0;
} /* end main */
```

The global **int** variables declared in Program 2 will be accessible to **all** functions defined in that program. The **main()** function has access to these global variables because of this feature of the language.

```
/*
 * Program 3 - PROG3.C
 */

#include <stdio.h>

int main (void)
{
/* local variables */
int a = 7;
int b = 2,
int c = 5;

    c = (a + b) / 2;
    printf ("a: %d b: %d c: %d\n", a, b, c);
    return 0;
} /* end main */
```

In Program 3, because **a**, **b**, and **c** were declared as local to **main()**, no other function within this program can access the values of these three variables. These three variables are shielded from influence by external functions.

**Useful Tip: Finding Global Variables**

Global variables are defined outside of functions.

## 3.9. Constants in C

Often, a programmer may want to assign a **constant** value into a variable. A constant is usually just a number on the right hand side of an equal sign, or a number that is part of a mathematical equation. In both PROG2.C and PROG3.C, we can see constants being assigned into integer variables prior to a mathematical operation being performed upon them.

For non-ANSI C compilers, there are two situations that require special handling of constants, to avoid a problem where all numeric constants were treated as integers, unless they were followed by specific modifiers.

For floating point constants, many non-ANSI C compilers require that the number be written as the whole number portion, a decimal point, and a fractional portion, even in situations where there is no fractional portion. For instance, the statements:

```
double x = 0.0;
. . .
x = 5;
```

may seem obvious to us that the floating point value 5 was being placed into the double precision floating point variable **x**. But older compilers would try to assign the integer 5 into the **double**, leading to an incorrect copy of two bytes into eight. To avoid such problems, the programmer would simply write:

```
x = 5.0;
```

for the assignment. To ensure no errors ever in floating point assignments or constant use, make sure that a floating point value has the decimal point, and at least a zero following if the number has no decimal digits.

For long integer constants, it is conceivable that you could run across a compiler that does not understand the following assignment:

```
long x = 0L;
. . .
x = 25321;
```

Older compilers would again try to assign an integer to the long integer variable **x**. To avoid this problem, again, a special notation was introduced:

```
x = 25321L;
```

By adding the letter L (uppercase or lowercase) to the end of the constant, we can be assured that the compiler will compile code to assign the long integer version of the number into the long integer variable.

In programming situations where constant values are altered many times during the development of software, or if the programmer wants to make a piece of code more readable, the following idea can be used:

```
#define CONSTANT_NAME      constantValue
```

The above generic syntax shows a constant being declared using the **#define** approach. This mechanism allows a programmer to create a name for a constant, and have the name associated with the constant's value. This idea *does not* allocate or create variables - it simply allows the programmer to use a name instead of a constant number.

For example, study the following:

```
#define CANADA_POPULATION  30000000L
. . .
long pop = CANADA_POPULATION;
```

The above code shows a definition of a constant called **CANADA_POPULATION**, and this long integer constant (value of 30000000) is being placed into a variable called **pop**. The **CANADA_POPULATION** constant can be used anywhere a programmer wants to reference a number that represents Canada's population, and makes the software much more readable as a result! These constants are usually named with a totally-uppercase name.

The same result can be achieved by using the **const** keyword preceding a variable declaration, like this:

```
const long kCanadaPopulation = 30000000L;
```

These constants are usually named like variables, except for being preceded by the letter 'k' (short for "konstant" (yes, I know)).

Of the two methods, the **const** method is considered to be the best, as it incorporates a desirable use of data type to make code more solid. However, it is not universally better than #define (more will be discussed about this when we cover arrays).

Constant definitions are generally always placed after the header file include statements and before the function prototypes. There is typically no taboo against "global constants", as they cannot be changed and, thus, do not have the same pitfalls as global variables.

For more detailed information on constants and their use, please see the section on *Rules about Variables*.

**Useful Tip: Constant Use**

It's best to use constants defined using #define or const rather than hard-coded. That way, if you need to change it, you change it in one place rather than many places.

**Useful Tip: Constant Naming**

If you use #define to create a constant, you can name it totally in UPPERCASE with underscores (_) separating the words.

## 3.10. Character Constants

For character constants, it is common to try to put the letters of the alphabet, numbers, spaces and punctuation into variables. C compilers allow you to directly assign such

characters by placing the character inside of single quotation marks. Study the following lines of code:

```
char a = 'A';
char b = ' ';
char c = '\n';
```

The first assignment places the letter A into variable **a**. The letter A is actually represented inside of memory using the **ASCII** (pronounced “ass-key”) equivalent of the letter. ASCII (American Standards Committee for Information Interchange) values are simply numbers which are common to all computers so that one computer can represent the same set of letters as any other computer. This is the way two different computers can exchange textual information without error.

The second example above shows how to assign a space character into a variable: simply place the space inside of the single quotes. The third example shows how to put a newline character (what causes the cursor to move down a line on the screen) into a variable. The newline character is represented by a backslash followed by the character 'n', all surrounded by single quotes.

# ASCII TABLE

| Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | [NULL] | 32 | 20 | [SPACE] | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 1 | [START OF HEADING] | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 2 | [START OF TEXT] | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 3 | [END OF TEXT] | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 4 | [END OF TRANSMISSION] | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 5 | [ENQUIRY] | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 6 | [ACKNOWLEDGE] | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 7 | [BELL] | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 8 | [BACKSPACE] | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 9 | [HORIZONTAL TAB] | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | A | [LINE FEED] | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | B | [VERTICAL TAB] | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | C | [FORM FEED] | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | D | [CARRIAGE RETURN] | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | E | [SHIFT OUT] | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | F | [SHIFT IN] | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | [DATA LINK ESCAPE] | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | [DEVICE CONTROL 1] | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | [DEVICE CONTROL 2] | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | [DEVICE CONTROL 3] | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | [DEVICE CONTROL 4] | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | [NEGATIVE ACKNOWLEDGE] | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | [SYNCHRONOUS IDLE] | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | [ENG OF TRANS. BLOCK] | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | [CANCEL] | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | [END OF MEDIUM] | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | [SUBSTITUTE] | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | [ESCAPE] | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | [FILE SEPARATOR] | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | \| |
| 29 | 1D | [GROUP SEPARATOR] | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | [RECORD SEPARATOR] | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | [UNIT SEPARATOR] | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | [DEL] |

(the above table is from http://fr.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange#mediaviewer/Fichier:ASCII-Table-wide.svg)

A programmer is able to use a character-based constant in their programming by putting the codes listed above inside of **single quotation** marks.

For example, if you wish to set a variable equal to the letter Q, the character constant is

```
'Q'
```

and for a beep (or bell), the character constant is

```
'\a'
```

The use of a two-character (or longer) long character sequence within single quotes and starting with a backslash defines **a single character**. This is a notation that is used for most of the codes below 32 (the space character). They are termed **control codes** and were used to control ASCII based terminals in the earlier days of computers. There are some useful codes in the first 32:

| Common Name | Character | Meaning |
|---|---|---|
| CR | '\r' | carriage return |
| LF | '\n' | newline |
| TAB | '\t' | TAB |
| BELL | '\a' | beeps |

The presence of '\x' indicates that there will be more characters that define a hexadecimal value (see the Appendix on Counting Systems for more information on hexadecimal) for the single character.

It is advisable to always use a character constant inside of single quotes rather than the number that represents the character. This is especially important when dealing with Unicode characters (which is beyond the scope of this course). If you are interested in finding out more about internationalization through Unicode, there is a good short video found at https://www.youtube.com/watch?v=MijmeoH9LT4 (if it's still there when you're reading this),

## 3.11. How to Output Variables

We have been exposed to the standard library function **printf()** as a means of printing textual information to the screen. This function is also capable of formatting our variables to the screen as well, giving rise to the name of this function (printf <==> print formatted).

The following lines of code illustrate how to use **printf()** to print values of variables:

```
int x = 10;
long y = 3L;     /* use L to indicate a long value */
char z = 5;      /* this can be a number or a character */
double w = 9.0;  /* append the decimal place */

printf ("x is equal to: %d\n", x);
printf ("y: %ld; z: %c\n", y, z);
printf ("w is equal to: %f\n", w);
```

Notice that each of the above **printf()** calls uses a percent (**%**) sign followed by one or more characters. The percent sign is used to inform **printf()** that we want to output one of the parameters following the quoted string of characters. For the first example, we have **%d**, telling **printf()** to format the second parameter, x, as a signed integer. The second example uses two percent formatting strings: **%ld** tells **printf()** to output the second (**y**) parameter as a long integer and **%c**, which tells **printf()** to output the third parameter (**z**) as a character. In the last example, we are using **%f** to print the double precision floating point variable **w** using floating point notation.

The following table describes some of the more common formatting codes:

| Specifier | Output | Example |
|---|---|---|
| d | Signed decimal integer | 392 |
| u | Unsigned decimal integer | 7235 |
| o | Unsigned octal | 610 |
| x | Unsigned hexadecimal integer | 7fa |
| X | Unsigned hexadecimal integer (uppercase) | 7FA |
| f | Decimal floating point, lowercase | 392.65 |
| F | Decimal floating point, uppercase | 392.65 |
| e | Scientific notation (mantissa/exponent), lowercase | 3.9265e+2 |
| E | Scientific notation (mantissa/exponent), uppercase | 3.9265E+2 |
| g | Use the shortest representation: %e or %f | 392.65 |
| G | Use the shortest representation: %E or %F | 392.65 |
| c | Character | a |
| s | String of characters | sample |
| p | Pointer address | b8000000 |
| % | A % followed by another % character will write a single % to the stream. | % |

By using the above formatting strings, any data type discussed thus far can be displayed on the screen using **printf()**.

It is interesting to note that you may print out the numeric, or ASCII, value of a **char** variable using **%d**, **%u**, or **%x**, and also print out the ASCII equivalent character of any integer by using **%c**.

You may have also noted that in our earlier examples of **printf()**, we had the pair of characters **\n** being used. The proper term for these pairs of characters is **escape sequence**, and is used to embed special characters within our strings that we normally could not represent or type from the keyboard. The most common of these is '\n', used to go to the start of the next line. There are others, though.

| Escape sequence | Meaning |
|---|---|
| \n | newline |
| \r | carriage return |
| \b | backspace |
| \x00, \x01, etc. | hexadecimal values |
| \t | tab |
| \" | double quote |
| \\ | backslash |

Notice that if we wanted to place a backslash within the string for **printf()**, we need to escape it itself using another backslash. It looks funny, but it is a valid escape sequence! As an example, if you wanted to print a string which sends a beep, a tab and a carriage return to the screen, the formatting string for printf would be:

```
printf ("beep: \a tab: \t carriage return: \r");
```

It should be apparent that escape sequences and variable formatting can be intermixed anywhere within the formatting string for **printf()**, making this function extremely versatile and powerful. In fact, many programmers like to think of the formatting string as a method of specifying how the output will look, including the "slots" where information will be formatted. The extra parameters will correspond to the slots that have been specified.

To make the formatting string just a bit more flexible, the programmer can specify a number in between the percent symbol and the formatting code to specify the **padding** of the format. For instance:

```
printf ("x is equal to: %4d\n", x);
```

will print the value of the variable **x**, but because there is a 4 between the **%** and the **d**, the variable will be padded out to fill 4 characters on the screen. **printf()** will pad such format strings with extra spaces in front of the number to fill it to the appropriate number of characters. Using such a method, a programmer can easily print columns of lined up numbers simply by requesting a specified width or padding for the numbers in the table! As well, by adding a zero in front of the number, you can pad a numeric format with leading zeros! The following:

```
printf ("x: %04d\n", x);
```

will output to the screen (if **x** is currently equal to 10):

```
x: 0010
```

Finally:

```
printf ("real number: %8.2f\n", a);
```

will format a floating point number, taking up eight spaces in total, with two of those characters coming after the decimal point. It should be apparent that this is a convenient method of printing currency (dollars and cents) values to the screen. All of these nifty features of **printf()** clearly make this function a true workhorse!

## 3.12. Common Beginner Errors

- printf() always has a format string. This format string usually starts with a " and ends with a ".
- Any formatting codes present in a printf() formatting string must have an expression for the value to which it corresponds. That expression (which could be as simple as a variable) must follow the format string, separated from the format string by a comma.
- Other common printf() error examples:
    - printf("%d, number");
    - printf("%d");
    - printf(%d, number);
    - printf("%d") number;
    - printf("number");
    - printf("%d", number, "\n");
    - printf("The number is ", number, "\n");

## 3.13. Summary

In this chapter, we learned about the various data types that a computer is designed to handle as well as the names C gives to these different data types. C contains equivalents to the byte, word and long word data types, **char**, **int** and **long**, their unsigned counterparts (by prepending the keyword **unsigned** in front of the data type in the declaration), as well as two different floating point data types, **float** and **double**.

The syntax for a variable declaration was introduced, and the naming rules for variables were found to be identical to the naming rules for a function, as shown earlier.

Next, a discussion concerning where to declare variables led to the definition of two classes of variables: **global** and **local**.

We also discussed the methods of declaring constants of different types, and we had a look at the ASCII table of values.

Finally, we looked at outputting text using **printf()**, how to format the output of variables, and how to print characters which are normally not accessible within the formatting string of **printf()**.

# Chapter 4. Operators

## 4.1. What is an Operator?

As a programmer, we know that once we've defined our variables for our program, we will want to try to do something with these variables. Perhaps we will want to add two variables together, and store the result in a third variable. This is an example of two different **operators** in use: **addition** and **assignment**. It is through the use of the different operators that the C language offers that meaningful work can be performed in our C programs.

Addition is the simplest one to understand since addition is the same in math or in C programming. Assignment is a bit harder to grasp. An assignment is simply telling the C compiler to store a value inside of a variable. In our example above, the third variable is assigned the result of adding our first two variables together.

From math, we know that there are many **mathematical operators** to use. We can subtract, multiply, and divide. You will find that C naturally handles the standard operators but forces the programmer to create functions which perform more complex operations.

C will also allow you to perform **bitwise** operations, which allow the programmer to manipulate the bits directly inside of a variable. As well, we will see the use of comparison operators that allow the programmer to test if a value is equal, greater than, not equal or less than, another value.

## 4.2. C Operators

Most of the commonly found operators in any computer language are found in C as well. As in math, C contains two classes of operators: **binary** operators and **unary** operators. Binary operators are used in between two values upon which the operator is to act upon. Addition (+) is a binary operator since it will act upon two values on either side of the addition sign. The negation operator (-) is a unary operator since its job is simply to negate the value that follows the operator.

## 4.3. Binary Operators

Here are some of the more common binary operators:

| Operator | Use |
|---|---|
| + | addition |
| - | subtraction |
| * | multiplication |
| / | division |
| % | modulo |
| && | logical-AND |
| \|\| | logical-OR |
| == | equality comparison |
| != | inequality comparison |
| < | less-than |
| > | greater-than |
| <= | less-than-or-equal-to |
| >= | greater-than-or-equal-to |
| = | assignment of a value to a variable |

There are also **bitwise** operators, which are explained in the Appendix on Counting Systems. It suffices to know that they are used to manipulate the **bit**s within the various data types we've been exposed to thus far. The same goes for the shift operators, since they are used to move bits side to side within data objects.

The **&&** and || operators work in terms of **boolean** logic, the logic of **true** or **false**. The **&&** operator will produce a boolean true result when both sides of the operator evaluate

to boolean true, and the || operator will evaluate to a boolean true if either one or the other, or both sides of the operator evaluate to a boolean true value. Otherwise, the result of either operation is boolean false. The comparator operators such as equality, inequality, greater than, less than, greater than or equal and less than or equal will produce a boolean result if the comparison succeeds (true) or fails (false). These boolean operators will be used extensively when we explore flow of control later.

A curious operator is present in the above table: **modulo**. This operator is used to perform a division between two **integer** values, and extract the **remainder** of the division. This is the complement of the division operator, which will, in the case of the **char**, **int** and **long** data types (and unsigned versions of these data types), produce just the **whole number result**. For instance:

*7 / 3 = 2.333333333*

With modulo division, we obtain the remainder of 1, which is one third (1/3) of the way to the next even division of a value by 3.

## 4.4. Unary Operators

| Operator | Use |
|---|---|
| - | negation |
| ! | boolean inverse (a.k.a. not) |

The negation operator should be a familiar operator (changing a value from positive to negative and vice versa). The exclamation mark is the boolean inverse operator, which flips a true result into a false result, and vice versa.

## 4.5. Operator Examples

Here are some examples of how to use the above operators:

```
x = 5;     /* assignment */
y = x * 3; /* multiplication and assignment */
z = y % 2; /* modulo and assignment */
q = x && y;     /* boolean AND and assignment */
a = -y;    /* unary negation and assignment */
b = a * x / y + z;    /* combinations! */
```

In looking at the above examples, it's possible that the one that you might find most obscure is the third one, involving the modulo operator. However, this particular example is extremely valuable, as it assigns z with a value of 0 if y is even and 1 if y is odd. It's nice to be able to determine this in a single line of code!

## 4.6. Precedence Rules

It should be noted that in C, just like in math, there is an order of operations that the compiler will follow in evaluating an expression. For instance, multiplication takes precedence over subtraction, meaning that a multiplication will be done first before any subtraction. As an example, look at the following statement:

```
a = b * c - d * e;
```

Because multiplication is more important than subtraction, the compiler will generate code that will multiply variable **b** with **c**, then multiply **d** with **e**, and finally subtract the second multiplication result from the first before assigning it to variable **a**.

The precedence rules for the operators we've seen so far can be summarized using the following table:

| **Operators** | **What They Do** |
|---|---|
| * / % | Multiplicative |
| + - | Additive |
| < > <= >= | Relational |
| == != | Equality |
| && | Logical-AND |
| \|\| | Logical-OR |
| = | Assignment |

Notice with the above table of precedence, the assignment operator will always occur last when the C compiler generates code. Also notice the order of precedence for the logical and comparator operators since this ordering makes writing comparison logic much simpler in the long run.

Of course, just like in math, you may override the precedence order at any time by using brackets. For instance:

```
a = b * (c - d) * e;
```

looks similar to our earlier example, but because of the brackets, the compiler will generate code to subtract d from c before multiplying that result with b and e.

Note that adding brackets does not make code less efficient, it simply allows the programmer to alter the order in which a statement will evaluate or to make the desired

order clearer. Note as well that the brackets around an expression operate differently than the brackets required for a function call. Study the following incorrect expression:

```
a = b (c - d) * e;
```

This statement is similar to our previous statement, but notice how the first multiplication sign is left out. In math, this would be a valid way of expressing an implied multiplication, but in C, the compiler would think that we are multiplying e with the return value of a function called b!

**Useful Tip: Brackets in Arithmetic**

Use brackets rather than relying on your (or someone else's) knowledge of precedence in all but the most trivial cases.

## 4.7. Assignment Operators

Another important concept is the **assignment** operator in C. An assignment operator is a binary operator that is used to give a value to a single variable. This may sound odd, but an example of an assignment operator will clear up any confusion quickly:

```
x = 5;
```

The above statement simply puts the integer 5 into the variable x.

Here's a somewhat less familiar assignment operator:

```
x += 5;
```

The above statement uses the addition assignment operator, which will take the current value of the variable **x** and add 5 to it, and store that result back in variable **x**. The addition assignment operator will increment a variable by some amount. The above statement is equivalent to:

```
x = x + 5;
```

There are also similar assignment-like operators for other binary operators. For example, -= behaves like += except that subtraction is done instead of addition. Similarly, *=, /=, %=, etc.

## 4.8. Increment and Decrement

The final two operators to look at are the increment and decrement operators. The increment and decrement operators are used to alter a variable's contents upwards or downwards by one in value. The operators are ++ for increment and -- for decrement. The following lines illustrate how to use these operators:

```
x++; /* x = x + 1; */
y--; /* y = y - 1; */
z = --x;   /* x = x - 1; z = x; */
a = x++;   /* a = x; x = x + 1; */
```

The increment and decrement operators should always be used by programmers whenever a change upwards or downwards by one is desired by a C programmer.

Note however that a compiler will order increments and decrements different depending on which side of the variable the operator is on! In the third example above, the decrement is on the left-hand side of the variable **x**. Hence, we first subtract one from the contents of **x**, and lastly assign the new value of **x** to variable **z**. This is called **pre-increment.**

In the fourth example, variable x has an increment is on the right hand side. Here, we first take the current value of **x** and assign it to **a** and lastly increment the value of **x**. This is called **post-increment**.

Both statements are similar in nature, but totally different results will come out of either statement.

## 4.9. Putting it Together

The following program, OPERATE.C, utilizes some of the concepts outlined in this chapter.

```
#include <stdio.h>

int main (void)
{
int a = 5;
int b = 10;
int c = 0;
long d = 100L;
long e = 200L;
long f = 0;
double g = 3.14159
double h = 20.4;
double i = 0.0;
double j = 0.0;

    c = a * b - 5 * (a + b);
    c++;
    a++;
    c -= a;
    printf ("a: %d b: %d c: %d\n", a, b, c);

    f = e % d;
    printf ("d: %ld e: %ld f: %ld\n", d, e, f);
    f = (e + 1) % d;
    printf ("new value of f: %ld\n", f);

    i = g * g + h * h;
    j = g * (g + h) * h;
    printf ("g: %f h: %f i: %f j: %f\n", g, h, i, j);

    return 0;
}    /* end main */
```

## 4.10. Common Beginner Errors

- Watch out for the operators that have both single-occurrence and double-occurrence uses (e.g. = and = =, & and &&, | and ||, < and <<, > and >>).
- Very important note if you're used to programming in another language: if you're used to being able to declare variables anywhere in a code block in a language like C++, you'll find that you can't do this in C. Your variables must be declared **before** any non-variable-declaration line.

## 4.11. Summary

In this chapter, we learned about the different operators that a programmer in the C language has at their disposal. The different operators allow code to be written which can manipulate variables to generate the results a programmer desires. The operators include the standard mathematical binary operators (addition, subtraction, multiplication and division), along with unary operators (negation, boolean inversion), logical operators (greater than, equality, inequality, less than, logical AND, logical OR).

We also had a look at the precedence order for C, and noticed that it was similar in nature to the order of operations in math. Finally, we had a brief look at the assignment operators and the increment and decrement operators.

# Chapter 5. Flow of Control

## 5.1. Introduction

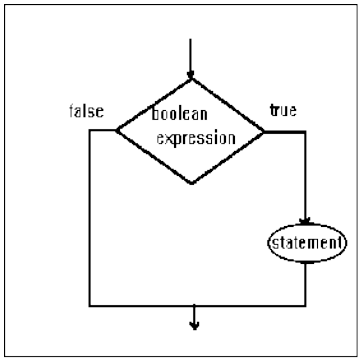
Through the course of the last few chapters, what should be apparent is that a function provides an implied feature: it temporarily causes your program to alter its **flow of control**, by stopping execution of the current function's code whenever a function is called (e.g. **printf()**). The function will then take over and execute, and when complete, it will return control back to where it was called. You can think of the flow of control of a program as the path a program may take, much like the path a driver may make on a long driving trip. You can get somewhere by driving along a highway (doing everything in the **main()** function), but you usually find more interesting and exciting things if you take a side road once or twice before getting back on the main highway. Likewise altering the flow of control within a program allows a programmer to produce, not only more interesting programs, but also more powerful and useful software.

Apart from calling functions, a C program has a number of other methods of altering the path a program may take. If you are familiar with other programming languages, you may be aware of certain concepts such as **looping**, as well as the ability of making a **decision** before proceeding to perform some task. These allow a programmer to offer a choice in how a program's path will be followed, thus creating the possibility of writing powerful, useful software with little effort. In this chapter we will begin to explore the methods the C programming language offers for changing the control flow within a program.

## 5.2. if Statement

As stated above, one method of altering how a program flows is through the use of a comparison, or decision. In C, as is the case in many other languages, a comparison is done by checking the **boolean**, or true/ false outcome of some expression, and if the result is **true**, some section of code will execute, and if **false**, some other section can be executed. The syntax for performing a decision in C is as follows:

```
if (boolean expression)
{
        statement;
}
```




if statement flowchart

Notice that for the **if** statement in C, the boolean expression to be tested **must** be within a pair of round brackets. The **if** statement in C looks much like a function call but should not be confused with one.

A very important note: an **if** statement is not programmed fully until a complete statement follows the closing bracket of the **if** portion of the statement. A complete **if** statement includes the boolean expression test, as well as a statement to be executed if the expression is true.

If the boolean expression turns out to be true, the **statement** following the closing round bracket will be executed. If false, the **statement** will **not** be executed, causing your program to move to the next line of code in your program.

**Useful Tip: Semicolons After if Statements**

Make sure that you don't overgeneralize and put a lone semicolon after the closing bracket for an if statement. The reason for this is that a semicolon is considered a statement that does nothing! Thus, the semicolon would be the body of code that is executed when the if statement's condition is true and the **real** body would always be executed, no matter what!

The following are a number of examples of valid comparisons in the C language:

```
if (x > 5)
{
    y = 3;
}

if ((x == y) && (a != b))
{
    z = 10;
}
```

The first example above does a check to see if the value of variable **x** is greater than the 5. If true, the statement **y = 3;** will be executed. If not, the statement **y = 3;** will be skipped, as if it was never in your program at all. This complete **if** statement tells the compiler that whenever the variable **x** contains a value greater than 5, the variable **y** will be assigned the value 3.

The second example is little more complicated. You should be aware that even though this example has a second line **z = 10;**, this line is still part of the same **if** statement. Recall that an **if** is not complete until you terminate it with a complete statement.

The lines of code between the curly braces following the if statement is called the **body** of the if statement. You must indent the body of an **if** statement by one tab stop to visually indicate what will execute if the **if** boolean expression is true. Well, really the compiler doesn't strictly require it but good programmers do :-).

**Course Requirement: Indentation After Decisions**

You must indent the body of the if statement to show that a decision has to be made and the indented code may or may not be executed. This also applies to while, do-while, and for. The principle is to indent the body after a decision. This also implies that the decision and body cannot be on the same line.

**Useful Tip: Spacing Around Brackets**

Use spaces near brackets. You can have one space before, one space after, or both. Being consistent with this will make your code look more regular and easier to understand. Being inconsistent will make you look amateurish.

## *5.3. Logical Operators*

You should also notice that this second example **if** statement is composed of two boolean expressions linked together using the **&&** binary operator, which is the **logical AND** operator. The way the logical AND operator works is that if the first operand evaluates to true, **and** the second operand evaluates to true, then the overall expression will be evaluated to true. If either one or the other expression evaluates to false, then the entire expression is false.

Above, the first operand for the binary logical AND is the boolean expression **x == y**. The operator == is the equality operator, which is another binary operator. The equality operator (usually read by C programmers as "is equivalent to" or "matches") instructs the C compiler to generate code to take the current value of the variable **x**, and compare it against the current value of the variable **y**. If they match, a true result is generated, if not, false is generated.

**Useful Tip: Confusion About = and = =**

Always remember that = does assignment and == compares equality. Unfortunately, it is very easy to put the wrong one in and not get any kind of compile error.

You will also run across this confusion with ||, &&, <<, >>.

You should be aware that the limitations of the font used in this document makes this operator look like two long lines, one above the other. Don't waste your time looking for the long lines on your keyboard ... it's two consecutive equal signs. Changing font to make it clearer gives us this: **==** .

The second operand in the example is the expression **a != b**. This expression is an inequality operation, (another binary operation) in which the value of variable **a** is compared to the value of variable **b**, and if they don't match, true is generated, otherwise false. The exclamation mark in C is the **not** symbol, and a C programmer normally reads **!=** as "not equal".

With the evaluation of both operands of the **&&** operator, the compiler will generate code to check these two conditions. As stated above, if both sides are true, the entire

expression evaluates as true, if not, the entire expression is false. If true, the statement **z = 10;** will be executed, otherwise, nothing will be executed.

As an exercise, let us assume the variables **a**, **b**, **x**, **y**, and **z** have the following values (the 2nd example is repeated for ease of discussion):

```
x = 20;
y = 20;
a = 10;
b = 10;
z = 99;

if ((x == y) && (a != b))
{
    z = 10;
}
```

The logic of the second example above would generate these boolean results:

**x == y** since **x** is 20 and **y** is 20, the result is **true**

**a != b** since **a** has the same value as **b**, and since we are checking for the condition of inequality, the result is **false**

**true && false** the overall result is false since both sides of the **&&** operator are not true results

since the overall result is false, the statement **z = 10;** is skipped, and z remains 99

Repeat the above exercise with the value of **b** set to 30. Prove to yourself that **z** will be set to the value 10 in this case.

## *5.4. Truth Table*

The following table may help you understand the operation of the logical operators, **&&** (logical AND), || (logical OR) and **!** (logical inversion).

| **Operator** | **Usage** | **Notes** |
|---|---|---|
| && | F && F == F<br>T && F == F<br>F && T == F<br>T && T == T | if expr1 is boolean true and expr2 is boolean true, then the entire expression is true; otherwise, it is false |
| \|\| | F \|\| F == F<br>T \|\| F == T<br>F \|\| T == T<br>T \|\| T == T | if either expr1 or expr2 are boolean true, then the entire expression is true; otherwise, it is false |
| ! | !F = = T<br>!T = = F | if expr is true, then generate false; otherwise, generate true |

In the table above, the letter T represents true, and the letter F represents false. This table of Ts and Fs is normally called a **truth table**, since it lists all the possible outcomes of boolean equations such as the ones we've see above.

It should be known that in C, if an expression evaluates as non-zero, the result will be considered true, and if the expression evaluates as zero, the result will be considered false. Thus, the following examples of **if** are valid:

```
if (x)
{
    y = 10;
}

if (!y)
{
    x = 10;
}
```

The first example above causes the variable **x** to be evaluated as a boolean expression. If **x** is equal to zero, this will generate a false result, and the rest of the **if** statement is ignored. If it is anything but zero, the expression will be evaluated as true, and the rest of the **if** statement will execute, setting **y** to 10.

The second example above checks to see if **y** is boolean false. In other words, if **y** is zero (false), **!y** will invert this false result into a true result, allowing the body of the **if** statement to be executed.

**if** statements such as the above are not uncommon since they are the short form ways of writing the following lines, which are equivalent to the above:

```
if (x != 0)
{
    y = 10;
}

if (y == 0)
{
    x = 10;
}
```

The longer form (involving the explicit comparison to 0) is definitely preferred, as it leaves no room for misinterpretation.

**Useful Tip: Boolean Comparisons of Non-Boolean Expressions**

Even though you see statements like `if (x)` every once in a while, you should avoid it since it can be confusing. It implies that x is a boolean variable when it usually is not. C++ supports a native `bool` data type. C does not.

You can combine more than one boolean expression together, using combinations of the logical operators, as well as using brackets to change the order of operations. For instance:

```
if (((x == 3) && (y != 2)) || (z < 5))
{
    printf ("wow!\n");
}
```

Notice how we are checking for the equality of variable **x** with 3 **and** the inequality of variable **y** with 2. If both expressions are true, then the logical AND expression is true. Notice, though, that our logical AND expression is logically ORed with the last expression, **z < 5**. If either the last expression is true, or the AND expression was true, then the entire logical expression is true, causing **wow!** to be printed to your screen. Tremendous sophistication can be achieved by using logical operators and brackets to select what sections of your code will be executed.

**Useful Tip: Code Clarity**

Consistency of style, proper use of brackets, and use of adequate whitespace to help with the clarity of your code is **always** more desirable than any decrease in source file size that may result from not doing so.

## *5.5. Putting It Together*

Now that we've seen how logical operators work in our software, let's have a look at an actual program which will use the **if** statement to perform a useful task. Note, to make this program useful, a number of function calls that are covered later are used to obtain user input.

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

int main (void)
{
char buffer[100] = {0}; /* used for user input */
int month = 0;
int day = 0;
int totalDays = 0;

    /* ask user to type in the month and convert */

    printf ("Enter month: ");
    gets (buffer);
    month = atoi (buffer);

    /* ask user to type in the day, and convert */
    printf ("Enter day: ");
    gets (buffer);
    day = atoi (buffer);
```

```
/* calculate elapsed days */
if (month > 11)
{
     totalDays += 30;
}

if (month > 10)
{
     totalDays += 31;
}

if (month > 9)
{
     totalDays += 30;
}

if (month > 8)
{
     totalDays += 31;
}

if (month > 7)
{
    totalDays += 31;
}

if (month > 6)
{
    totalDays += 30;
}

if (month > 5)
{
    totalDays += 31;
}

if (month > 4)
{
    totalDays += 30;
}

if (month > 3)
{
    totalDays += 31;
}

if (month > 2)
{
    totalDays += 28;
}
```

```
    if (month > 1)
    {
        totalDays += 31;
    }
    totalDays += day;

    printf ("Month: %d Day: %d Total Days: %d\n",
    month, day, totalDays);

    return 0;
}    /* end main */
```

**Useful Tip: Getting user input**

We're introducing some concepts this example that we'll be re-introducing later. Here's what you just have to trust me on for now:

- Getting a string from the keyboard can be done by calling the gets() function.

- Strings can be stored in a variable like this: char buffer[100]

- Converting a string to an integer can be done by passing the string to the atoi() function.

This is the largest program that we have studied at to date but it's also a rather useful program. If you've ever wondered how many days have elapsed since the start of the year, this program will calculate it.

This program is interesting in its use of the **if** statement to help us determine how many days to add to the running total of days from the start of the year.

First off, the computer will ask you to enter both the month and day to work with. Both should be a simple string of numeric digits (ie: 10 for October), and the program will convert the user's input into the appropriate data type (in this case, an integer). After conversion, the integer is stored in the variables **month** and **day**.

Next, the program will begin checking to see if the month specified is greater than a specific month, and if so, it will increment the running total by the number of days in the previous month. Finally, after running through the entire series of **if** statements, the day will be added to the running total, to give an accurate assessment of the number of days elapsed since January 1 of the year.

The above logic sounds rather convoluted at first, but it is actually quite simple if you think about why we are adding the total number of days from the previous month.

Let's use an example of March 16. March is month 3. After setting the running count to 0, we now begin to follow the series of **if** statements. The first **if** statement is:

```
if (month > 11)
{
    totalDays += 30;
}
```

Here we check to see if **month** is greater than November (month 11). If **month** was 12 (December), we would have added 30 to the running count. Since we are dealing with March, this **if** is false, and we leave **totalDays** alone.

The next **if** is encountered, and again, nothing changes since March is not greater than October.

This continues until we come across:

```
if (month > 2)
{
    totalDays += 28;
}
```

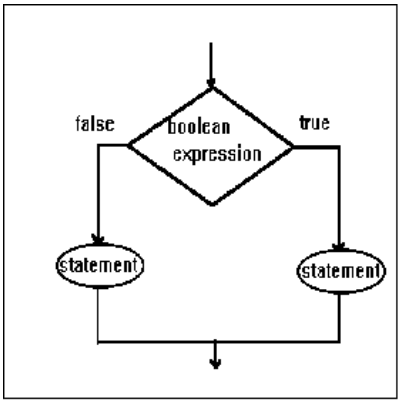
This statement is asking if the **month** is greater than February. In our case, March is indeed greater than February (3 is greater than 2), and because the comparison is true, we increment **totalDays** by 28 days (not taking into account leap years - we'll leave that for a student exercise!).

The next statement asks if **month** is greater than January, and again, March is greater (3 is greater than 1), and we again increment the counter, this time with 31.

What we've actually done here is add the total number of days in January, with the total number of days in February, since those are the only **completed** months for the date March 16. We still need to add the 16 days in March to our count, which is done by:

```
totalDays += day;
```

to arrive at the total number of days since the start of the year.




**if else** flowchart

Try working through on paper other examples of dates to prove to yourself that this logic does indeed correctly compute the number of days from January 1.

## 5.6. Chained if Statements

The next topic concerning **if** statements is the ability to **chain** multiple if statements one after another. This is made possible by the fact that the C programming language allows the programmer to specify what to do if the result of the boolean expression is false. This is done using the **else** statement. The syntax for a fully exploited **if** statement is as follows:

```
if (boolean expression)
{
    statement1;
}
else
{
    statement2;
}
```

If the boolean expression is true, **statement1** is executed, otherwise (**else**) **statement2** is executed. Now we can do two different things based upon the true or false outcome of the boolean expression.

The idea of chaining multiple if statements comes in handy when doing things like comparing user input against a series of expected options. For instance, imagine a program which gives the user a choice of four different options to perform. Once the user has entered the choice, a check is made against the expected input values, and if they match, the appropriate option is taken. Otherwise, some sort of error is displayed.

The following example uses a series of chained if statements to perform such an operation:

```
if (key == '1')
{
    performOptionOne();
}
else if (key == '2')
{
    performOptionTwo();
}
else if (key == '3')
{
    performOptionThree();
}
else if (key == '4')
{
    performOptionFour();
}
else
{
    printf ("ERROR: Enter number from 1 to 4!\n");
}
```

Notice that a chained **if** statement is simply an **if** performed as part of the **else** portion of the previous **if**. In the example above, if the key entered is not the key labeled '1', then a check will be made against key '2', and so on. If it was key '1', then option one will be performed, and the rest of the **if** statements will be ignored (since they are only executed when the first **if** fails!) Therefore, when the keystroke is indeed '1', the other 3 tests will **not** be done as there is no need to, and the chained series of **if** statements ensures that this principle is adhered to.

**Useful Tip: no elseif**

There is no such thing as **elseif** in C. You must separate the **else** from the **if.**

## 5.7. Compound Statements

There are often times a C programmer wishes to do more than one statement of code if the result of an **if** is true or false. We've so far seen that an **if** statement in C will only execute the complete statement following the closing round bracket of the **if** statement (or the **else** portion of an **if else**).

In C, a programmer has the ability to create something called a **compound statement**, which is simply a series of statements which fall between an opening brace (**{**) and a closing brace (**}**). The statements in between them are considered to belong to the same **block of code**, and C will treat such compound statements as a single entity for the purposes of an **if** statement or the **else** statement. For example:

```
if (x < 10)
{
     printf ("x is less than 10!\n");
     printf ("setting y equal to 20\n");
     y = 20;
}
else
{
     printf ("x is not less than 10!\n");
     printf ("setting y equal to 0\n");
     y = 0;
}  /* endif */
```

The above example shows two compound statements in action: the first compound statement prints two messages to the screen and sets the variable **y** equal to 20. The second compound statement also prints two (different) messages to the screen, and assigns the value 0 to **y**. If the boolean expression **x < 20** is true, the first compound statement will be executed fully (the two **printf()** function calls and the assignment of 20 into **y**). If false, the second compound statement is executed fully.

Notice again the **style** being used for both bracing and spacing: this is a common compound statement style used in professional C programming, as mentioned earlier.

With compound statements, a programmer has the freedom of being able to do as much or as little as is required for a particular programming problem.

Even though the compiler doesn't force you to put braces around a single statement that is used as a body (like was done earlier), it is a good idea to do so. That way, if you add a second statement to the body, you don't have to remember to put braces around the newly-compounded body.

**Course Requirement: Braces on Single Statement Bodies**

You are required to put braces on single-statement bodies in this course.

## 5.8. Nesting if Statements

The final topic for **if** statements is the ability to **nest if** statements. The term "nest" is used to denote the action of "doing an **if** inside of another **if**". An example will make this idea immediately obvious:

```
if (y > 20)
{
     if (x < 10)
     {
          printf ("x < 10 and y > 20");
     }
     else
     {
          printf ("x > 10 and y > 20");
     }
}/* endif */
```

Here we have a compound statement, and inside of the compound statement we have nested another **if else** statement. If the first **if** statement finds **y** is greater than 20, then the second **if** statement within the compound statement will be executed, and based upon the value of **x**, one of two messages will be printed to the screen. This is yet another example of the freedom and flexibility of C.

## 5.9. Loops

So far we have looked at the abilities of C when it comes to making comparisons within our programs. There are many circumstances, however, where simply making a decision will not always allow a program to perform the desired result. For instance, if you wrote a program to prompt the user to enter two numbers, and the program multiplied these numbers and printed the answer, the programmer may want the user to try again. At this moment, we have no method of being able to perform the action of "trying again".

The first thought some novice programmers have is: "why not just call **main()** again before **main()** is finished?" Although it is technically permitted, this situation leads to an interesting programming phenomenon called **recursion**, which is the term given to the calling of a function from within itself. It sounds like a simple enough idea, but the mechanism the microprocessor goes through when a function is called causes some of your computer's memory to be used up temporarily until the function is completed, and repeated calling of yourself begins to eat up more and more memory, until there isn't any left. Improperly programmed recursive situations are generally fatal as far as getting stable and robust software to be generated.

**Course Requirement: Recursive Calls of main()**

It is typically a very bad idea to recursively call **main().** You will lose marks if you do so.

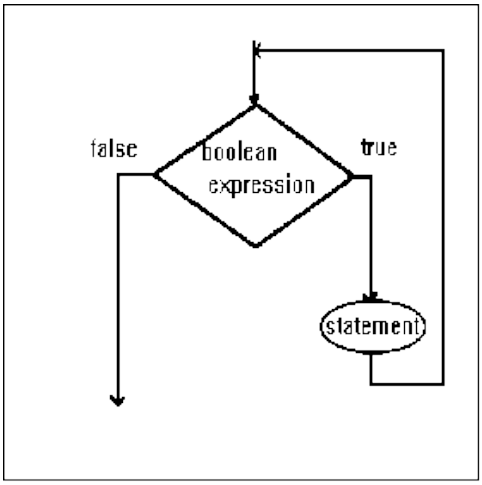
We are still left with a dilemma: how do we "try again"? The answer lies in the fact that C, as well as most other computing languages, allows the program to specify where, and how often, a block of code is to **loop**. With a loop of some sort, a programmer can instruct the computer to execute a piece of code 10 times, 100 times, or even forever.

In C, there are three main methods of performing a loop: the **while** statement, the **do-while** statement, and the **for** statement. We will now have a closer look at the **while** statement now.

## *5.10. The while Statement*

The **while** statement is a looping mechanism in C which appears remarkably similar to the **if** statement upon first glance. The syntax of the **while** statement is as follows:

```
while (boolean expression)
   statement;
```




**while** statement flowchart

If the boolean expression inside of the brackets of the **while** statement evaluates to true, **statement** will be executed. Once **statement** is complete, the **while** statement will perform another check of the boolean expression. As long as the boolean expression is true, **statement** will be executed over and over and over. Therefore, the **while** statement can be thought of as a simple **if** statement that has the ability to evaluate the expression multiple times until the expression is false. Note that the **statement** above can be a full blown compound statement just like in the case of an **if** statement. Note as well that **statement** is commonly referred to as the **loop body**.

Let's have a look at a simple example using the **while** statement:

```
#include <stdio.h>

int main (void)
{
int x = 0;

    while (x < 10)
    {
        printf ("The value of x is: %d\n", x);
        x++;
    }/* end while */

    printf ("DONE\n");
    return 0;
}/* end main */
```

In the above example, we initialize a variable **x** with the value of zero (remember initial values can be given to variables as they are declared). We then begin the **while** statement, checking to see if **x** is less than 10. Currently, zero is indeed less than 10, and because the boolean expression result is true, we enter the compound statement making up the body of the loop.

First we print a message informing the user of the current value of **x**. Next, we increment **x** by one. The compound statement is now complete, and the **while** statement will again check the current value of **x** against 10. **x** should now be 1, thus this boolean expression is still true, and we enter the loop body a second time.

This loop body will in fact be executed 10 times, printing the following on our computer screen:

```
The value of x is: 0
The value of x is: 1
The value of x is: 2
The value of x is: 3
The value of x is: 4
The value of x is: 5
The value of x is: 6
The value of x is: 7
The value of x is: 8
The value of x is: 9
DONE!
```

Notice that the value of 10 is **not** printed on the screen. This is because we are expected to execute the body of the loop as long as **x** is **less than** 10. When **x** is equal to 10, (and hence, no longer less than 10) the loop will terminate, causing the message **DONE!** to appear on the screen. To force the loop to count to 10, use the <= operator in the boolean expression.

A typical usage of the **while** statement is to handle the “try again” type of situation. For example, study the following code snippet:

```
int done = 0
int key = 0;

. . .

done = 0;
while (done == 0)
{
     . . .
     if (key == 'Q')    /* check for quit key */
     {
          done = 1;
     }
}    /* end while */
```

The **while** statement checks to see if the variable **done** is boolean false, and as long as this variable stays false, the loop will execute. This type of loop is very common in C programming.

Inside the loop we check, using an **if** statement, a keystroke pressed by the user. If the keystroke is equal to the letter 'Q', the program will set the variable **done** to 1, which is true in boolean. As soon as **done** is true, the loop will terminate (you can think of the fact that **done** is true as literally meaning that we are done this loop!).

## 5.11. Pitfalls of Flow of Control

Now that we've seen the various flow of control constructs C provides, it is time to look at some pitfalls that a programmer may come across when writing software with such statements.

Study the following statement:

```
if (x == 3) ;
```

Remember the previous Useful Tip about the semicolons? Well, the fact is that an **if** statement is not complete unless it has the boolean expression as well as a statement and semicolon. The above example appears to be missing a statement to execute if the expression is true. Actually, as far as the compiler is concerned, there is indeed a statement to execute, and that statement happens to be ... **nothing**! This is an example of an **empty** statement: a statement that has no code associated with it. If you wrote this line into your program, the compiler would generate code to perform the test, and if the expression were true, nothing would happen! It may appear odd in this case, but it is a common coding error that occurs when working with flow control statements such as the **if** statement. Avoid the temptation of putting a semicolon everywhere in your code – remember, terminate a statement **only** when it requires termination, and that should allow you to avoid the above pitfall.

A common problem that novice C programmers run across when programming a loop is the inadvertent placement of a semicolon after the closing round bracket of a **while** or **for** statement (and the **if** statement as shown above). As another example, study the following statements:

```
x = 0;
while (x < 10);
{
         printf ("x: %d\n", x);
         x++;
}    /* end while */
```

The above loop is never ending! Can you spot why? Note the semicolon after the closing round bracket of the **while**. We've just told the compiler to generate a loop that will execute a **null** statement as long as the value of **x** is less than 10. Since just prior to the loop we initialized **x** with a value of zero, (which is certainly less than 10), this loop will never stop looping, as it never has a chance to move **x** towards 10! Luckily, some compilers today detect that such a situation exists, and a warning will be issued, giving you an indication that something is amiss with your source code.

Another common mistake is to program a loop whose keep-going condition cannot possibly be met during the execution of the loop body. Study the following example:

```
y = 30;
z = 0;
x = 0;
while (x < 10)
{
    z += x * y;
    y++;
}
```

The intent of the above loop is to sum the product of **x** (from zero to 10) and **y** and store the result in a running counter **z**. But the error occurs in how we've programmed the **while** statement that specifies how we reach termination. Notice that we've stated to increment **y** by one each time through the loop. But our keep-going condition says to break the loop when **x** reaches 10. In this case, **x** will never reach 10, and we've got another endless loop!

Many endless loop problems occur due to typing mistakes by the programmer. Seldom is there a need to write a program with an endless loop. Because of that, when most compilers detect that condition, a warning is generated to inform you of this potentially dangerous situation.

## 5.12. Indentation Revisited

**Three bracing styles with if statements, this time**

**Style 1: Braces level with code block**

```
if( i < 8 )
{
    i = 8;
}
```

**Style 2: Braces level with control statement**

```
if( i < 8 )
{
    i = 8;
}
```

**Style 3: Braces attached to control statement (K&R Style)**

```
if( i < 8 ) {
    i = 8;
}
```

Choose one of these styles and stick with it. If you need to experiment, keep all code in a project consistent and change on the next project. Chances are, though, that one style will seem natural to you and the others will seem alien :-).

## 5.13. Common Beginner Errors

- Avoid putting a single = in an equality comparison.
- Avoid putting that nefarious semicolon at the end of the **if** or **while** statement, unless you really know that it should be there. If it should be there, put it on a separate line so that it's very obviously not a mistake.
- Put braces around single-line bodies to avoid problems later.
- When joining conditions using && and ||, make sure that you have chosen the right one (it's easy to get confused).

## 5.14. Summary

This chapter has greatly expanded our knowledge of the C programming language with the introduction of **control flow** statements. These types of statement allow the programmer to choose which portions of code will be executed at a given time, through the use of comparisons and possibly loops as well.

C uses the **if** statement to accomplish the act of performing a comparison. This statement will evaluate the boolean outcome of the expression between the brackets of the **if** statement, and if it results in a boolean true outcome, the rest of the **if** statement will be executed.

We learned that a C programmer has the ability to cause a program to loop until a certain condition is met. This is accomplished using the **while**, **do ... while** and **for** statements.

Care must be exercised to avoid coding situations where an **infinite** loop may occur - a loop that has no way of terminating itself.

# Chapter 6. Pseudocode

At this point, you should be thinking that our programs will be getting to be more complex. There is a temptation to just sit down in front of a computer and start writing code without doing some preliminary design. Except in the most trivial situations, this is a bad idea. Either you'll waste time by having to do and redo your code or you'll write bad, inflexible code that will cause you to be cursed by your co-workers.

Luckily, there are many methods that you can use to avoid your co-workers' wrath. One of the simplest to understand and implement involves using **pseudocode**. Pseudocode is an English-language representation of what you have to do with your program to solve the problem that you've been given. It should be understood that the name "pseudocode" indicates what it is and what it is not: it **is** something that **looks like** code but it is **not** something that **is** code. You can write pseudocode to solve a problem without knowing anything about a computer language. As a matter of fact, if you think of it as just describing how you do something (in English), you know how to do it!

Here's an example of simple pseudocode:

```
Loop while there is input from the file.
    Process the input line.
    Print the resulting output.
End loop.
```

There are six basic operations in pseudocode (from *Simple Program Design*, pages 10 to 12):

- A computer can receive information.
- A computer can put out information.
- A computer can perform arithmetic.
- A computer can assign a value to a variable or memory location.
- A computer can compare two variables and select one of two alternative actions.
- A computer can repeat a group of actions.

This leads to three basic control structures:

- Sequence
- Selection
- Repetition

## *6.1. Sequence*

Sequence refers simply to having executable statements in a row, one executed after another. There are no decisions to make and no looping. This describes the types of programs that we've worked with so far.

## 6.2. Selection

Selection refers to making a decision. Once the decision is made, some subsequent code is executed if the decision result is True. If the result is False, the aforementioned code is not executed.

## 6.3. Repetition

Repetition refers to repeating a block of code over and over again. A decision is made, either at the top or bottom of the loop, determining whether or not the block is executed. If not, the next line of code after the block (and decision, if at the bottom) is executed.

## 6.4. Example of Pseudocode

Here is an example of pseudocode for the Rock, Paper, Scissors game (from *Programming Logic and Design* by Joyce Farrell, Course Technology, Boston, 2004, ISBN 0-619-16021-7, page 53):

```
Start
     Ask a friend to play a game of Rock, Paper, Scissors with you
     While answer is yes
          Extend yourHand and myHand
          If yourHand is "paper" then
               If myHand is "rock" then
                    winner = yourHand
               Else
                    If myHand is "scissors" then
                         winner = myHand
                    Else
                         winner = tie
                    Endif
               Endif
          Else
               If yourHand is "scissors" then
                    If yourHand is "rock" then
                         winner = myHand
                    Else
                         If myHand is "paper" then
                              winner = yourHand
                         Else
                              winner = tie
                         Endif
                    Endif
               Else
                    If myHand is "rock" then
                         winner = tie
                    Else
                         If myHand is "paper" then
                              winner = myHand
                         Else
```

```
                        winner = yourHand
                    Endif
                Endif
            Endif
        Endif
        Ask a friend to play a game of rock, paper, scissors
    End while
Stop
```

In this example, we can see many examples of all three types of pseudocode.

**Useful Tip: General program structure**

Here's an example of general program structure:

1. Do stuff that happens at the end of the program, including declaring and initializing variables.

2. Perform the main loop.

3. Do stuff that happens at the end of the program.

## 6.5. Why Pseudocode?

It is well established that time spent in pre-coding design will be rewarded in time efficiency when producing software. If you design your software with emphasis on what it does rather than the details of how the language works, your success will be much greater!

The next chapter will give some examples of working with pseudocode with flow of control situations. For more detail on using pseudocode, read *Simple Program Design.*

## 6.6. Common Beginner Errors

- Always design your software before coding it!!!!!!

## 6.7. Summary

This chapter has introduced the concept of using pseudocode for design.

# Chapter 7. Designing for Flow Of Control

## *7.1. Some Help With Flow of Control*

So far, the code that we've had to deal with doesn't require a lot of analysis to produce. Flow of control issues, though, make you think a bit more. If you're experienced with doing flow of control in other languages, skip this chapter. If not, though, take a look at Chapters 4 and 5 in *Simple Program Design* (by Lesley Anne Robertson, Course Technology, Cambridge, MA, 2000, ISBN: 0-619-01590-X).

## *7.2. More General Flow Of Control Design Principles*

When analyzing a problem, watch for any indications of actions that result from a decision being made. That usually indicates the need for an **if** or **while** statement. Sometimes there isn't an explicit decision that is obvious but there is an indication that there are multiple items being processed in some way. This usually indicates the need for a loop (e.g. **while**). The next section will contain some examples of translating problems into pseudocode.

## *7.3. Some General Flow Of Control Design Examples*

### Example 1

#### Problem Statement

Write a program that prints the sum of prices entered by the user.

#### Commentary

There are two indicators of a loop here: **sum** and **prices.** The fact that prices is plural indicates that you are dealing with more than one item. Also, a sum of only one item is trivial, so it should be assumed that we're summing more than one.

If we were told exactly how many items we were dealing with, we wouldn't necessarily require a loop. Since we don't know, though, we need a loop to handle it.

#### Solution:

- Initialize sum to 0.
- Loop while there are prices to get.
  - Get a price.
  - Add the price to the sum.
- End loop.
- Print sum.

## Example 2

### Problem Statement

Write a program that prints "PST Taxable" if a restaurant meal is above $3.99 and "No PST" otherwise.

### Commentary

Here we see the word "if". This is a sure giveaway that we have to make a decision. This decision is expressed as an if-else construct.

### Solution

- Get price.
- If price is above 3.99, then
  - Print "PST Taxable".
- Else
  - Print "No PST".
- End If.

## Example 3

### Problem Statement

Write a program that gets restaurant meal prices from the user, printing "PST Taxable" if the price is above $3.99 and printing "No PST" otherwise.

### Commentary

Using the first example, we can see that "prices" indicates a loop. We also have a decision, indicated by the use of the word "if". We've got to determine whether the **if** is within the **while** or vice versa. The giveaway here is that the decision is on a single price, not on all of them collectively. Thus, the decision is made on a single price as we're processing it within the loop.

### Solution

- Loop while there are prices to get.
  - Get a price.
  - If price is above 3.99, then
    - Print "PST Taxable".
  - Else
    - Print "No PST".
  - End If.
- End Loop.

## Example 4

### Problem Statement

Write a program that gets restaurant meal prices from the user, printing "PST Taxable" if the sum is above $3.99 and "No PST" otherwise.

### Commentary

Here, we're making the decision on the sum. That indicates that the decision could be outside the loop, after we've determined the final sum. A slight modification to the problem would change the solution, though, as we'll see in the next example.

### Solution

- Initialize sum to 0.
- Loop while there are prices to get.
  - Get a price.
  - Add the price to the sum.
- End loop.
- If sum is above 3.99, then
  - Print "PST Taxable".
- Else
  - Print "No PST".
- End If.

## Example 5

### Problem Statement

Write a program that gets restaurant meal prices from the user, printing "PST Taxable" as soon as the sum is above $3.99 and "No PST" if it never is.

### Commentary

We've got to be careful with this one, since we're working with the sum, which might indicate that we need a decision after the loop. The difference, however, is that we're not dealing with the **final** sum but instead with an interim sum (or **running total**). The interim sum occurs within the loop and only becomes the final sum after exiting the loop. Thus, we've got to make that decision inside the loop.

The statement of the problem is subtle, though, in that it says "as soon as the sum is above $3.99". That indicates that we do not want to print the message more than once. To facilitate that, we will create another variable called **over399** that is 0 if the sum is not over $3.99 and 1 otherwise.

We can't forget the second part of the problem, though. We have to print "No PST" if the sum is never above $3.99. The only time that we know that is after the loop is complete. So, we've got to make another decision after the loop, as well.

### Solution

- Initialize sum to 0.
- Initialize over399 to 0.
- Loop while there are prices to get.
  - Get a price.
  - Add the price to the sum.
  - If sum is above 3.99 and over399 is 0, then
    - Print "PST Taxable".
    - Set over399 to 1.
  - End If.
- End loop.
- If over399 is 0, then
  - Print "No PST".
- End If.

## Example 6

### Problem Statement

Write a program that prints the sum of restaurant meal prices, adding 6% GST to all meals and 8% PST to individual meals above $3.99. The sum should be printed as the total of the meals and all taxes.

### Commentary

We've got a sum so we've got a loop. We have one decision to make: whether or not to apply PST. Since we're doing this on individual meal prices, we have to do it inside the loop. We should note that we do not have to decide whether or not to apply GST. It's always applied.

### Solution #1

- Initialize sum to 0.
- Loop while there are prices to get.
  - Get a price.
  - Add the price to the sum.
  - Calculate GST from the price.
  - Add GST to the sum.
  - If price is above 3.99, then
    - Calculate PST from the price.
    - Add PST to the sum.
  - End If.
- End loop.
- Print sum.

### Solution #2

- Initialize sumNoPST to 0.
- Initialize sumPSTApplied to 0.
- Initialize totalMeals to 0.
- Loop while there are prices to get.
  - Get a price.
  - If price is greater than 3.99, then
    - Add the price to sumPSTApplied.
  - Else
    - Add the price to sumNoPST.
  - End If.
- End loop.
- Calculate PST by multiplying sumPSTApplied by the 8% PST rate.
- Calculate total_meals by adding sumPSTApplied and sumNoPST.
- Calculate GST by multiplying totalMeals by the 6% GST rate.
- Print sum of totalMeals, PST, and GST.

## Post-design Exercise

Write the C code that corresponds with the pseudocode in the examples above.

## More Examples

For more examples, read Chapter 6 in *Simple Program Design.*

# Chapter 8. Writing Your Own Functions

## 8.1. Review of Functions

Earlier, we briefly covered the essential concepts regarding **functions**. A function is the way a C program is constructed: indeed, without the function called **main()**, a C program could not exist.

A C program, when properly written, will employ many small functions that carry out specific tasks. These functions break up the **flow of control**, allowing sub-functions to execute, alleviating the **main()** function from doing all the work. In this chapter, we will look at reasons for building our own functions, and the next chapter will look at some of the simpler **library** functions available through our C compiler.

## 8.2. Why Write a Function?

This is a common question beginner C programmers will ask, as a C program will faithfully execute every line of code you may have inside of your **main()** function without ever a need to call an external function.

One of the most important reasons is that a good C programmer will always try to make code as **modular** as possible, localizing important or repeated computations and procedures within easily maintainable and accessible functions. By creating functions to do such tasks, we not only reduce the chance of programmer error in typing in repeated chunks of code, but we also make it simpler to adjust the operation of a task. By simply changing a single function within our program, rather than perhaps five or six occurrences of a similar piece of code, we can eliminate much of the maintenance effort in software development.

Here's an example of a typical repeated computation:

```
/*
 * SECONDS1.C
 */

#include <stdio.h>

int main (void)
{
long hours = 22L;
long minutes = 52L;
long seconds = 11L;
long totalSeconds1 = 0;
long totalSeconds2 = 0;


    totalSeconds1 = hours * 60L * 60L + minutes * 60L
         + seconds;
```

```
    hours = 10L; minutes = 20L; seconds = 30L;

    totalSeconds2 = hours * 60L * 60L + minutes * 60L
        + seconds;

    printf ("Elapsed time: %ld\n", totalSeconds1 -
        totalSeconds2);
    return 0;
}    /* end main */
```

The example above shows **main()** declared to return an **int** data type. The **main()** function has also been declared to accept no parameters, due to the **void** parameter list. This means that **main()** needs no input information for it to operate.

The **main()** function above is attempting to compute the number of elapsed seconds between two sets of times. The computation that is repeated is to convert the hours, minutes and seconds into the number of elapsed seconds from midnight using the formula **h * 60 * 60** + **m * 60** + **s** (60 minutes in an hour, 60 seconds in a minute). This computation is repeated twice inside of our program. Because of the dual repetition, there is twice the chance for typing error, or other such problem that can cause our program to have a "headache". If we wished to repeat this calculation for 30 different times, it will become exceedingly tedious to enter this formula over and over again.

Note: in the above program, the use of the capital **L** after the constants is done since when working with long integers, it is usually a good idea to specify exactly to the compiler that the constant is a **long** constant as well. Recall that older, non-ANSI C compilers would do funny things if you did not specify the L after a constant if assigning it to a long integer.

## *8.3. Creating a Function*

Study the following rewrite of **SECONDS1.C**. You should notice the creation of a programmer defined function called **calcSec()** which is called with parameters, much like **printf()** to calculate the elapsed time.

```
/*
 * SECONDS2.C
 */

#include <stdio.h>

/* the definition of a new function */

long calcSec (long h, long m, long s)
{
long sec = 0;

    sec = h * 60L * 60L + m * 60L + s;
    return sec;
}    /* end calcSec */
```

```
int main (void)
{
long totalSeconds1 = 0;
long totalSeconds2 = 0;

    totalSeconds1 = calcSec (22L, 52L, 11L);
    totalSeconds2 = calcSec (10L, 20L, 30L);
    printf ("Elapsed Time: %ld\n", totalSeconds1 -
        totalSeconds2);
    return 0;
}    /* end main */
```

The function we have created is called twice to calculate the elapsed seconds from midnight for two different sets of values. Notice that the computation was coded only once inside of the function called **calcSec()**, and we called this function twice to perform this calculation.

## 8.4. Function Declaration Notation

As we saw in chapter 2, a function is declared using the following notation:

```
return-data-type name-of-function (parameter-list)
```

In **SECONDS2.C** the function **calcSec()** has been declared to return a **long** integer to the caller of the function, and the parameter list specifies that three pieces of information will be given to this function, the three **long** integers: the hour **h**, the minute **m** and the second **s**. **calcSec()** will take these input parameters and calculate the number of elapsed seconds since midnight using the exact same formula as in **SECONDS1.C**.

Notice that inside of the body of **calcSec()** we are assigning the results of the computation into a variable called **sec**. This variable is declared to be a **long** integer within the braces defining the body of **calcSec()**. Recall that this is one of two locations we can declare variables, and this form of variable is referred to as a **local variable**. The term "local" is used to identify the nature of this type of variable: the variable will only be in existence as long as the function is executing. As soon as the function is complete, this variable will disappear from existence until the next time the function is called. This characteristic is called the **lifetime** of the variable. It is sometimes beneficial to think of a local variable as a **temporary** variable, needed only as long as a piece of code is executing.

In **SECONDS2.C**, the local **sec** is created as soon as we begin execution of **calcSec()**, and will be removed from memory as soon as **calcSec()** is complete. Again, think of local variables as temporary variables to hold intermediate computations of your functions.

## 8.5. Returning a Value

**calcSec()** takes the parameters given to it (**h**, **m** and **s**), computes the number of elapsed seconds and stores the result in **sec**. We then use the statement

```
return sec;
```

to tell the C compiler that we are ready to return a value back to the caller of this function.

The above concept is vital to the understanding of the usefulness of functions in C. Rather than doing the computation directly in the **main()** function as in **SECONDS1.C**, we have the function do the computation based upon the parameters specified, and the function will give the result back to **main()** to be stored in the appropriate variables.

It may be instructional at this point to think of a function as a method of performing some operation, and the operation it performs may generate a result. If we ever needed to calculate the number of seconds that have elapsed since midnight, we now have a function that can compute this result, and the result can be obtained by calling **calcSec()** with the appropriate parameters.

The return value can be thought of as a "special variable" in the CPU, and whenever the **return** statement is used to return a piece of information, this "special variable" will be assigned the value you wish to return. Imagine that **return sec;** performing code such as:

```
specialVariable = sec;
```

The caller of the function has the option of either using or ignoring the return value. In our example here, the caller (the **main()** function) is choosing to use the return value.

Notice how we are assigning the return value of **calcSec()** into two different long integer variables. Using our "special variable" analogy, these long integer variables are obtaining new values from this "special variable", and we can now use these values in further computations.

## *8.6. Parameters*

In the above discussion, we briefly talked about passing parameters to the function **calcSec()**. Parameters can be thought of as pieces of input your function can use in order for it to carry out its work. For instance, the **printf()** function uses a parameter which is normally a series of characters in between double quotation marks. When using **printf()** to print formatted variables in our programs, we add more parameters after the quoted string of characters, all separated with commas, to match the formatting codes in the string. The **printf()** function is special in the world of C programming since it is capable of accepting a variable number of parameters when called. Most functions we use in C require the programmer to supply the correct number of parameters for the function to succeed in doing its job, and this goes for programmer defined functions as well.

Parameters for a function are actually local variables that just happen to be initialized with values at the time the function is called. Parameter variables are therefore simply copies of the information supplied when the function gets called.

For **calcSec()**, when it is called in the **main()** function, we are supplying three different pieces of information, the hour, minute and second. Notice that both times that **calcSec()** is called, we are supplying a set of 3 long integer constants as parameters. When **calcSec()** begins to execute, it will copy the values that are supplied at the time of the call in **main()** into the parameter variables **h**, **m** and **s**.

Input parameters for a function do not need to be constants. They can be the result of an operation, a variable or a constant. Parameters for a function can even be the return value of another function! Remember that a parameter is simply a local variable that happens to have its value initialized when the function is called. Have a look at the following example program cel2fahr.c that gives an example of other types of parameters for a function that converts Celsius to Fahrenheit.

```
/*
 * cel2fahr.c
 *
 * Convert Celsius to Fahrenheit
 */

#include <stdio.h>

double celsiusToFahrenheit (double c)
{
    return (c * 9.0 / 5.0 + 32.0);
}/* end celsiusToFahrenheit() */

int main (void)
{
double c = 0.0;
double f = 0.0;
double a = 0.0;
double b = 0.0;

    f = celsiusToFahrenheit (10.0);
    printf ("C: %f F: %f\n", 10.0, f);

    c = 15.0;
    f = celsiusToFahrenheit (c);
    printf ("C: %f F: %f\n", c, f);

    a = 5.0;
    b = 10.0;
    f = celsiusToFahrenheit (a * b);
    printf ("C: %f F: %f\n", a * b, f);

    f = celsiusToFahrenheit (celsiusToFahrenheit (20.0));
    printf ("C: %f F: %f\n", celsiusToFahrenheit (20.0), f);
    return 0;
}/* end main */
```

The main feature of this program is the function **celsiusToFahrenheit()**, which accepts a single parameter called **c**, of type **double**, and will use that parameter to compute the Fahrenheit equivalent for that Celsius temperature. Once computed, it will return the Fahrenheit back to the caller as a **double**. Notice that this time, we wrote the function

without using a local variable to store the result. This is because all the function is supposed to do is compute a result and return it. In simple situations like this, there is really no need to allocate a local variable since all we would be doing is returning the value stored in that local variable.

In the **main()** function, we are calling **celsiusToFahrenheit()** in many different manners, each using a different method of specifying a parameter for **celsiusToFahrenheit()** to use. The first call to **celsiusToFahrenheit()** uses a double precision constant as the parameter. Note that to specify a double precision constant, a C programmer normally will place a decimal point and then at least a zero or other decimal digits. Recall that older, non-ANSI C compilers sometimes have trouble with floating point constants when there are no decimal digits, as some compilers will default them to an integer, much like the long integer problem earlier.

After receiving a copy of the specified parameter, the function will faithfully convert that Celsius value into a fahrenheit value, return that value, and then we print out the values using **printf()**.

The next example uses the double precision floating point variable **c** as the parameter. We are able to tell **celsiusToFahrenheit()** to convert the value that **c** holds into fahrenheit. Please note that the parameter **c** in the function **celsiusToFahrenheit()** is in a totally different piece of memory than the local variable **c** in the function **main()**. The parameter in **celsiusToFahrenheit()** is simply a **copy** of the variable of **c** in **main()**, just like it's a copy of the constant in the previous example.

The next example uses two variables, **a** and **b**, each set to their respective values, and multiplies them as the parameter to the function **celsiusToFahrenheit()**. After the multiplication is completed, the result (in this case 50.0) will be given to **celsiusToFahrenheit()** as the parameter. Even the result of a computation can be used as a parameter.

The final example is the use of the return value of **celsiusToFahrenheit()** itself as a parameter **celsiusToFahrenheit()**. We convert 20.0 into fahrenheit, and then use that conversion as the next value to convert! It may be a rather pointless exercise, but it does show the power of functions and their parameters: function return values can be used as parameters themselves, offering absolute flexibility when writing code to perform some task.

## *8.7. Function Comments*

A style issue has to do with the description of what a function does and what it needs to do it. This is typically handled using a **function comment**. A function comment describes the name of the function, the purpose of the function, the parameters passed and return values for the function (if any).

An example for the previous function is:

```
/*
 * Function: celsiusToFahrenheit()
 * Description: Converts a Celsius temperature to a Fahrenheit
temperature.
 * Parameters: double c: temperature in Celsius
 * Return Values: temperature in Fahrenheit
 */

double celsiusToFahrenheit (double c)
{
    return (c * 9.0 / 5.0 + 32.0);
}    /* end celsiusToFahrenheit */
```

This may seem rather trivial but, believe me, you will write more complex functions that will require more complex comments. Another example, without the function contents:

```
/*
 * Function: parseTicketRequest()
 * Description: Takes a string that contains a request for
 * a ticket for a flight and parses it.  The string should
 * have the following format (separated by spaces):
 *   originatingAirport destinationAirport flightNumber
 *   month day year
 * The two airports are three character long airport codes.
 * The month, day, and year are all integers.  If all
 * fields are present and correct, 1 is returned and the
 * values are filled in to the parameters.  Otherwise, 0 is
 * returned and the contents of the parameters are
 * undefined.
 * Parameters: char *input: input string
 *         char *orig: originating airport code
 *         char *dest: destination airport code
 *         int  *flight: pointer to flight number
 *         int  *month: pointer to month of flight
 *         int  *day: pointer to day of flight
 *         int  *year: pointer to year of flight
 * Return Values:    1 if all fields are correct;
 *              0 otherwise
 */
```

Now, you can't possibly tell me that it is easier to determine that information present in the preceding function comment by looking at the code!

**Useful Tip: main() Function Comment**

You don't need a function comment for main() because it's covered by the description within the header comment.

**Useful Tip: Function Comment Veracity**

Of course, it is possible for function comments to become outdated. Make sure that you change your function comments when the function changes to make the comment incorrect. Also, take a quick look comparing the function comment with the contents when you take over possession of an unfamiliar function for the first time.

**Course Requirement: Function Comments**

You must have function comments for all functions you create except main(). This comment must be immediately before the function definition.

## *8.8. Prototyping Functions*

With the advent of the ANSI C standard, compiler makers have been adhering to a principle adopted by the standards committee, and this same standard is used by (and strictly enforced by) C++ compilers. The standard is in regards to how a C compiler should interpret a function call prior to the function having been defined.

Imagine a **main()** function which wants to execute a function called **foo()**. If the function **foo()** is not declared before the **main()** function in your source code, as has been the case in the earlier examples in this chapter, the compiler will have to make assumptions about what the function will return.

All C compilers have been written to assume that a function, which has not been formally defined prior to its first call in a C source code file, will return an **int**. In fact, if a function does not have its return type specified, the compiler will default it to an **int** for you! This may be a simple way of saving some typing time for a programmer, but it is a dangerous programming practice, especially when you call a function that returns something other than an **int**.

It is sometimes possible for a C programmer can structure the source code to place all functions to be called ahead of **main()** so that the compiler has been able to compile these functions, and know what the return type of the function is. In this way, there will be no ambiguity in compilation. But what about cases such as the following (note that this program has an error purposely embedded to demonstrate prototyping):

```
/*
 * FOO.C
 */

#include <stdio.h>

/* here's where the prototypes should go! */

long foo (void)
{
    return (test(50));
}/* end foo */

long bar (void)
{
    return (foo());
}/* end bar */

int test (int a)
{
    long x = 0;

    if (a < 10)
        x = foo();
    return (x * bar());
}/* end test */

int main (void)
{
    long a = 0L;

    a = test (7);
    printf ("a: %ld\n", a);
    return 0;
}/* end main */
```

Although FOO.C is quite cooked in its operation, it nonetheless exhibits the problem we've been discussing. Notice how we've defined functions **foo()**, **bar()** and **test()** ahead of the **main()** function, as we've done in all our other examples. The problem lies in the fact that the function **foo()** ends up calling function **test()**, so, following our earlier approach, we should have placed **test()** in front of **foo()**. But looking inside of **test()**, we notice that we end up calling **foo()** based upon the outcome of a conditional test. Because we are calling functions prior to them being declared, the compiler will, in this example, think that the function **test()** returns only an integer, which will cause problems further down the road in our software.

This problem is not unlike the old chicken and egg problem - which comes first? The solution that ANSI C supplies for us is the use of a **function prototype**. The function prototype is a replica of the function declaration, except for the fact that the function

body is **not** declared along with the prototype. The main purpose of the prototype is to declare to the C compiler the return values of the function. As well, it defines ahead of time the data types of the parameters. Armed with this knowledge, the C compiler would be able to perform a number of useful checks for us.

- it can warn us when we have a return data type mismatch (when we try to return a data type that the function was not declared to return)
- it can check to ensure the data we pass are of the correct data type that the function expects.

This ability makes ANSI C compilers a real boon to programmers since the compiler can do a lot of the error checking for us.

The prototypes for the functions in FOO.C are as follows:

```
long foo (void);
long bar (void);
long test (int);
```

Note that the prototype almost exactly matches the syntax for a function declaration, except for formally declaring the names of parameters, and actually defining the function body. The use of the data type **void** as the parameter of some of the above functions is simply an indication to the compiler that, under no circumstances, shall those functions accept a parameter. This again allows the compiler to perform more error checking on your code.

A function prototype should be placed in the source code before the first attempt at using that function. If not, the entire reason for using this error checking mechanism will be defeated!

**Useful Tip #1: Placing Prototypes**

Most programmers will try to place their prototypes before any actual function declarations, and after the header files are included in your program. In the example FOO.C, this location is where the comment **/* here's where the prototypes should go */** lies.

**Useful Tip #2: Making Prototypes**

You can have parameter names in the parameter list for the prototype, even though they aren't used. This means that the easiest way to create a prototype would be to copy and paste the actual function definition, putting a semicolon at the end of the bracket closing the parameter list.

**Course Requirement: Prototyping**

You must prototype all functions that you write except main() (which has its own implicit prototype). You must also provide **void** in the parameter list if there are no parameters (this is different from C++). Failing to do so will lose you marks.

It should be noted that the header files you include, such as **stdio.h**, or **stdlib.h**, usually include their own function prototypes. This is because the standard library functions that

the C compiler manufacturer supplies need to be prototyped just like a programmer defined function. Because of this, we can be assured that we will never obtain an erroneous result back from a function simply because the compiler assumed a different return type that was anticipated.

## 8.9. Functions That Do Not Return Any Values

There are many circumstances where a programmer may wish to write a program that does not return a value to the caller of that function. Such functions, of course, will be declared to return **void**, meaning that no information will be given back to the caller.

Situations where this may arise are numerous. For example, envision a program which presents a menu to the user. The menu itself may be programmed to be displayed using a function called **displayMenu()**. This function has the job of showing the text of the menu on the screen, and nothing more. Once all the text is shown, the function will terminate, and nothing will be given back to the caller of **displayMenu()**.

As the example above shows, functions that return void simply perform some sequence of statements. This type of function is usually termed a **procedure** in most other programming languages, to differentiate it from a **function** that will return a value. In the C language, both functions and procedures are one and the same; only the return type differs.

## 8.10. Designing Using Functions

Now we have to determine exactly how we should use functions. A common method is that you start with a statement of what a solution to a problem does and you break it down in steps (typically using pseudocode). At each line of pseudocode within each step, you ask yourself **"Do I know how to do this easily?"** If the answer is "yes", then you can continue on to the next line. If the answer is "no", then you have to break that line down into other lines that describe what that line means or does. Each of the lines that has to be broken down can represent a function call.

### Partial Example:

Design a program that allows a student to make their own timetable, given established course times. It will be a menu-based program that has the following menus:

*Main Menu:*

1. Display my current timetable.
2. Display a course's times.
3. Quit.

*Current Timetable Menu:*

<stuff representing your timetable>

1. Delete course.
2. Return to main menu.

*Course Menu:*

<stuff representing your timetable>

<more stuff representing course times>

1. Show course.
2. Add shown course section.
3. Return to main menu.

*Quit Menu:*

1. Save changes.
2. Discard changes.
3. Return to main menu.

Assume that you have easy access to the student's current timetable and to the course scheduling info through provided function calls.

## Step 1:

```
Loop
    Display Main Menu.
    Do Action.
Repeat until done.
Quit
```

## Commentary:

The above is a rather simple statement of a solution to the problem. If you ask yourself the question **"Do I know how to do this easily?"**, it should be pretty obvious that you would answer "yes" to the Loop and Repeat lines but "no" to the Display, Do, and Quit lines. Thus, you would break Display Main Menu, Do Action, and Quit down further in Step 2.

This would also likely be an indication that the Display Main Menu and Do Action items should represent functions within our program.

## Step 2: Breakdown of *Display Main Menu*

```
Display Main Menu:
    Display Title
    Display each menu line
    Prompt user
```

## Commentary:

We start with a label that indicates what we are breaking down (Display Menu Item). This label could correspond to a function name. Within the body itself, we could

probably answer "yes" to the first two lines but "no" to the prompting of the user. Thus, the prompting gets broken down in another step (probably as another function).

## Step 2: Breakdown of *Do Action*

```
Do Action:
     Choose action
     Do specific action
```

## Commentary:

This is pretty simple. We do the action by first determining what we need to do and then doing that specific thing. Both can be functions that can be broken down further.

## Step 2: Breakdown of *Quit*

```
Quit:
     <fill in here>
```

## Commentary:

We don't know the details so we'll leave this out of our example.

## Step 3: Breakdown of *Do Specific Action*

```
Do Specific Action:
     Did they select item #1?
     If so, then do item #1
     Etc.
     Did they select something else
     If so, print error message
```

## Commentary:

This particular breakdown is only one of many ways that we could do it. We would also have to break down each individual action in another step.

## Step 3: Breakdown of *Prompt User*

```
Prompt User:
     Print prompt
     Increase cursor flash rate
     Beep annoyingly if they take too long
```

## General Commentary:

It should be obvious that this could go on for a long time, with many steps and much breaking down of functions. It's much better to put the work in here than having to redesign your solution later when you discover that it won't work the way it is supposed to.

**Useful Tip: Always Design First**

You should **always** design your solution "on paper" first. Of course, it doesn't have to be physically done on paper; you could do it in a text editor using pseudocode. For any non-trivial program, though, going straight to coding is almost always A Bad Idea.

**Useful Tip: The Magic Question**

At each line of pseudocode within each step, ask yourself **"Do I know how to do this easily?"**

## 8.11. Naming Conventions, Revisited

Here's a naming convention you should use:

- Start function names with a lowercase letter.
- Start variable names with a lowercase letter.
- Start macros with an uppercase letter.
- Put constants defined using #define in UPPERCASE, using underscores to separate words (e.g. #define SIZE 10).
- Make constants defined using const start with the letter k (e.g. const int kSize = 10;).
- Make function and variable names expressive and understandable.

**Useful Tip: Naming Conventions**

Always be consistent in your naming. It will save you tons of time in debugging!

**Course Requirement: Naming Standards**

Take a look at the SET Coding Standards to see what is required.

## 8.12. Common Beginner Errors

- Always have a main().
- Always have prototypes. When you change your function definition, change your function prototype at the same time.
- Don't define a function within another function. Function definitions must be separate from one another.
- Don't put a semicolon at the end of your function definition.
- **Always design before coding!!!!!!**

## 8.13. Summary

In this chapter we learned about the concepts needed to enable a programmer to define their own **functions**. A function can be used to perform often-repeated programming tasks, as well as making a program modular, allowing much simpler maintenance later

down the road. We also looked at how to declare and use the return value of a function, as well as how to declare and use the parameters for a function.

We also looked at the concepts regarding function prototyping, which allow a C compiler to be informed of the return value data type as well as the data types of the parameters for a function. An ANSI C compiler can then perform a large amount of error checking for us.

Finally, we looked at the need for designing before coding. Functions tend to fall easily out of designing on paper, making your life easier.

# Chapter 9. Rules about Variables

It is useful at this point in our study to look at the proper rules regarding how variables are allocated and accessed in the C language. Because C is a compiled language, there must be no ambiguity about how memory is accessed, or where it will be allocated.

## *9.1. Scope*

The scope of a C variable is defined as "from where" a C variable can be used from. The declaration of a variable or function is active over a certain region of the source code text, which is known as its scope. The scope of a variable can be thought of as where you can access a certain variable, and where you can't. For functions, it can be thought of as when the C compiler knows the return type of a function, and the exact parameter lists for the functions.

In this text, we have seen two styles of C variables: **global** and **local**. If you recall, a global variable is a variable that is defined outside of a function declaration, and can be accessed by **all** functions within the C program. Certain programming constructs become quite easy to master because of this feature: for example, a globally defined character buffer for string input is not only useful, but desirous in some circumstances since you do not usually want multiple instances of a large input buffer eating up the memory space your program has to work with. A table of values, such as the number of days in each month, is another example of a useful global variable.

A local variable, on the other hand, is a variable that is declared internally to a function. This class of variable includes parameters for a function, as well as the locals declared within a function. The big advantage locals have over globals is that they will **never** conflict with your global variables even if they have the same name. A local variable called **counter** will never interfere with a global variable called **counter**. If you have one hundred functions inside of your C program, and you declare a local variable for each of these 100 functions called **counter**, none of those 100 instances of **counter** will affect the globally defined variable **counter**, or each other for that matter.

To study the concepts behind scope in C, let us use the following program, which declares a number of globals, and uses them:

```
/*
 * SCOPE.C
 */

#include <stdio.h>
#include <stdlib.h> // for atoi()

int a = 0;
int b = 0;
char buffer[100] = {0};

int test (void);/* prototype */
```

```
int main (void)
{
     long c = 0L;

     printf ("Enter a number: ");
     gets (buffer);
     a = 5;
     b = atoi (buffer);
     c = (long)a * (long)b;
     printf ("a: %d b: %d c: %ld\n", a, b, c);
     return 0;
}/* end main */

int test (void)
{
     int c = 0;

     c = a * b;
     return c;
}/* end test */
```

The above program declares a number of globals, and inside of the two functions **main()** and **test()**, a local variable by the name of **c** is being declared. The **c** in **test()** will never conflict with the **c** in **main()** because of the scope rules in effect: a local is hidden to all functions except the one that is currently executing.

**Useful Tip: Naming Repeated Variables**

It is a good practice to use the same variable names for local variables across different functions. If you think about it, it wouldn't make much sense to have a counter represented by variables names i, j, count, counter, n, x, and y in different functions (since they don't overlap and interfere with each other).

The scope characteristics for C variables are fairly simple:

- Globals are accessible throughout your program at any time. These variables are created as your program starts and the memory occupied by these variables is freed when the program terminates.
- Locals are only accessible as long as your C program is executing a function that has that local variable defined. These variables are created as your program begins a function (or any block of code, for that matter) in which the variable is declared. As soon as that function (or code block) completes, the local variable ceases to exist.
- A local variable with the same name as a global variable has precedence over the global variable.

## 9.2. Lifetime Rules

Along with the concept of scope comes the idea of the **lifetime** of a variable. The lifetime of an object is defined as ”the period of time during which the object is allocated storage”. There are a number of classes of lifetime as well.

Firstly, **static** lifetime refers to variables that exist as long as the program is running. All global variables are static lifetime variables since they are allocated memory when the program begins and retain that memory until the program terminates.

Next, **local** or **automatic** lifetime refers to variables that are created when a compound statement begins and are destroyed when a compound statement ends. All local variables and parameters to a function are created as local lifetime variables, as well as variables defined at the start of a compound statement such as a **for** or **while** loop. As an example:

```
for (x = 0; x < 5; x++)
{
int y = 0;

    y = x * x;
    printf ("x: %d x squared: %d\n", x, y);
}    /* end for */
```

The variable **y** is created inside of the open and close braces for the **for** statement, which is its lifetime. The scope of this variable is only between those braces as well, thus **y** is invisible and inaccessible outside of the **for** statement. **y** is created when the **for** statement begins to loop, and is destroyed when the **for** statement is complete. This is exactly how local variables for a function operate.

## 9.3. Static Variables

A local can be forced to be a static variable by simply pre-pending the type modifier **static** in front of the type declaration. For example:

```
void foo (void)
{
static char x = 0;
    . . .
}    /* end foo */
```

declares a statically allocated byte of memory called **x** which exists throughout the entire time your program runs, but is only accessible during the time its scope rules are in effect.

A static local, although given memory at the time your program begins, is still only accessible when the function that declared it is executing. The advantage of this is that you can save the results of local computations in such statics, and only have particular functions access them rather than all functions!

Often a programmer may want to count how often particular functions are being called. Rather than declare a number of global variables such as **countFunction1**, **countFunction2**, and so on, the programmer can simply declare a number of static local

variables, all with the name **count**. Because they are local, even static locals with identical names will never conflict because of the scope rules for local variables.

Global variables and functions too can be declared to be static, and the usefulness of this stems from the fact that the C compiler will make the lifetime of such a function or global valid only inside of that source code module. You can declare a static global **x** inside of **test.c**, and that global **x** will only be useable inside **test.c**. If you linked **test.c** with **foo.c**, and **foo.c** had it's own static global **x**, you will find that there are two separately accessible global variables called **x** inside of that program!

## 9.4. The Processor Stack

It should be noted here that local lifetime variables are created on the **stack** of the CPU executing the program. The stack is a section of memory set aside where values are placed and taken off of in a stacking method: whatever is placed (**pushed**) on the stack last will be the object that is removed (**popped**) first, much like making a stack of magazines on a table: the last one placed on top is the first one removed. The typical scenario that is followed by a C program when a function call is made is given by this pseudocode:

```
push the last parameter onto the stack
push the second last parameter on the stack
. . .
push the first parameter on the stack
push the code address of the next line of C code
         to be executed after the function is complete
call the function

the function will create a frame pointer to point to where
         the stack variables are currently stored
         (the parameters)
the function will move the stack pointer to allocate enough
         space for the local variables

the function now begins
. . . (code for function)
the function now ends
the stack pointer is returned to where the frame pointer
    was pointing (clearing out the locals)
the return address is unstacked, and jump there


we are now at the line following the function call:
    the C program must unstack all parameters to
    "clean" the stack after the function call
```

It is obvious that a rather large amount of work is accomplished by a simple call to a function. This is the overhead of a function call, as alluded to earlier in this chapter. You

can see how the lifetime rules are utilized by the above scenario: the local variables are created on the processor stack each and every time the function is called, and when the function is complete, the stack is cleared out and the variables disappear until the next invocation of the function.

## 9.5. Memory Layout

As a final note, a brief discussion of a typical C program's memory layout is in order:

```
high memory
        --------------------------------
        end of heap

        . . .

        start of heap
        --------------------------------
        start of stack

        . . .

        end of stack
        --------------------------------
        end of global/static variables

        . . .

        start of global/static variables
        --------------------------------
        end of code

        . . .

        start of code
        --------------------------------
low memory
```

In the diagram above, memory grows in address from low memory (address 0) to high memory (address dependent on the computer system). Thus, the start of code is lower in memory (in address values) than the end of code.

The term **heap** refers to memory reserved for **dynamic** allocation of data types and structures, to be covered in a more advanced text. It should be noted that as your program uses more and more heap space, less memory is available for dynamic allocation of data. This is not really a concern for small C programs, but is of great importance to programs such as database managers that must handle the heap space carefully to maximize the amount of memory available for data inside of the database.

The **stack** is a curious entity - it is used to house the return addresses and local variables for the program - but if you notice in the above diagram, the stack is 'upside down' compared to the rest of the memory segments. Stacks grow 'downwards' in memory, due to how the internal architecture of stack handling is designed in virtually **all** microprocessors.

**Useful Tip: Preserving Stack Space**

The stack is usually a lot smaller than other memory areas. Make local array variables static if they are large in order to avoid using up valuable stack space.

## 9.6. Common Beginner Errors

- Don't give local variables the same names as global variables.
- Don't use inconsistent naming styles for variables (e.g. cursor_x and cursorX). You might end up with two variables with similar names while thinking that you've got only one.

## 9.7. Summary

We then had a discussion about the **scope** rules and **lifetime** rules of different classes of variables in C. Scope defines where a variable will be accessible, and lifetime defines when variables will be created and destroyed.

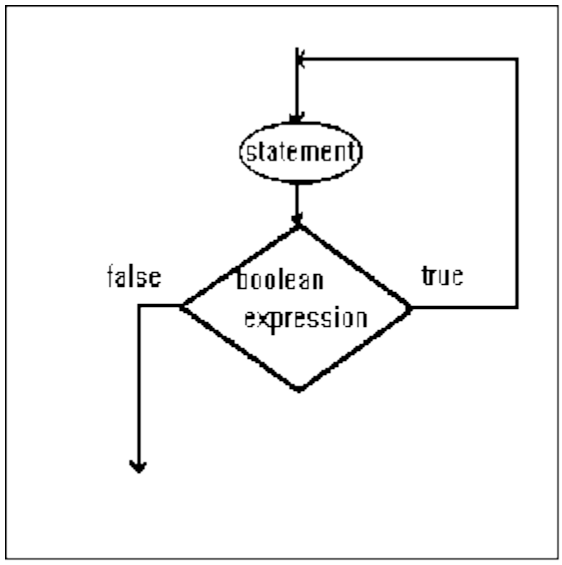
Finally, we took a brief look at what exactly happens when a function call is made on a typical microprocessor. This helps explain some of the scope and lifetime rules discussed above. A look was also made of the typical memory model of a microprocessor-based computer, which should give a programmer a better feeling of how memory is utilized in the computer.

# Chapter 10. Other Control Statements

## *10.1. do - while Statement*

The **do - while** statement is very similar in nature to the normal **while** statement, except it always allows at least **one** iteration (run) through the loop body **before** performing a check on the boolean expression. The syntax for a **do - while** statement is as follows:

```
do
    statement
while (expression);
```




do ... while flowchart

Note that the **do** and **while** keywords are separated by the loop body. The loop body will continue to be executed as long as the expression evaluates as boolean true. Again, it should be stressed that with this form of the **while** loop, the check of the boolean expression is done at the end of the loop, thus allowing at least one run through the loop. Depending on the particular programming problem, this form of the **while** loop may be more efficient than the standard **while** loop. Most other programming languages refer to this looping construct as a **repeat until** loop, since you will repeat (or do) the loop body until (or while) some condition causes termination.

**Useful Tip: Repeat-until**

The difference between a do-while and a repeat-until is that the conditions are reversed.

The following is a rewrite of the **while** example from the previous chapter, showing how the same thing could be accomplished using a **do - while** statement:

```
int done = 0;
int key = 0;
. . .
do
```

```
{
    . . .
    /* check if quitting */
    if (key == 'Q')
    {
        done = 1;
    }
    else
    {
        done = 0;
    }
} while (done == 0);
```

Notice that this time, we did not set **done** ahead of the loop since there is no need to "fudge" the condition as the condition is not checked until the **end** of the body of the loop. Since in both cases we wanted to execute the loop body at least once, the second example is a somewhat better approach for the "try again" type situations.

**Useful Tip: Bracing do-while**

No matter what bracing style you use, it is acceptible to put the while on the same line as the closing brace.

**Useful Tip: do-while Semicolons**

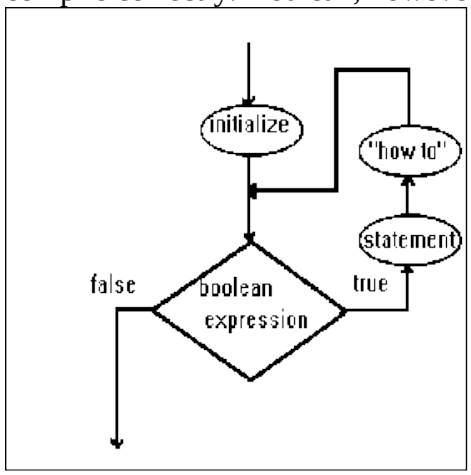
Make sure you don't forget the semicolon closing the do-while.

## 10.2. The for Statement

The final looping statement we will study is the **for** statement. This is the most complex of the three looping constructs, but it is one of the most powerful. The syntax of the **for** statement is as follows:

```
for (initialization; boolean-condition; how-to-get-there)
        statement;
```

Notice that for the **for** statement, there are three distinct parts within the round brackets: an initialization section, a boolean condition section, and a section which specifies how we are to arrive at a termination situation. The three sections are all separated using semicolons, and if any of the two semicolons are missing, the **for** statement will not compile correctly. You can, however, leave out any of the sections.




for statement flowchart

Let's have a look at a typical **for** statement, and dissect its operation:

```
for (x = 0; x < 10; x++)
{
        printf ("value of x: %d\n", x);
}
```

In the above **for** statement, we can see the three sections of the **for** in use. Firstly, we initialize the value of the variable **x** to zero. The initialization instruction within the **for** statement is executed only once at the start of the loop, thus **x** will be set to zero only once.

Next we have the boolean condition. The C compiler will generate code that will cause the loop body to execute as long as this condition is not false. In our example above, this **for** statement will loop as long as the value of **x** is less than 10. As soon as it is 10 or above, this condition is false, and we stop the loop. This condition will be checked exactly once prior to each execution of the loop body. The first check is done as soon as the initialization is complete, thus a **for** statement may not always execute the loop body. Some programmers like to refer to this boolean condition as a **termination** condition,

because as soon as it is false, the loop will terminate. However, it should be noted that the actual statement of the condition doesn't indicate when the loop should **terminate** but instead when it should **not terminate**. Thus, you might want to refer to it as the **keep-going condition** instead.

The last part of the **for** statement is the specification of what you change to stop the loop. In the above example, we will reach the termination of the loop by incrementing **x** by one each iteration (repeat of the loop). This increment will be done exactly once after each run through the loop body, and just prior to the next check of the boolean condition. The above loop will execute the **printf()** statement exactly 10 times, printing the values of **x** from 0 through 9 inclusive, since when **x** is equal to 10, this will break the boolean condition.

**Useful Tip: Semicolons After Other Control Statements**

Remember the useful tip about the semicolon and the if statement (that is, don't put a semicolon after an if!)? Well, it applies to while, do-while, and for as well.

NOTE: That doesn't mean that you should never put a semicolon after one of those statements. It simply means that you should only put it in if you truly want the program to do nothing if the condition is true.

**Useful Tip: Difference between C and C++ with for statements**

Unlike C++, you cannot declare a variable in the for statement itself in C.

## *10.3. Putting It Together*

The following program, called MULT.C will ask the user to enter which times table they want a printout for, and a **for** loop will be employed to generate the table for values from 0 through 12. Please note it uses some new functions, and new data objects that we haven't yet seen, but will be discussed in the next chapters.

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

int main (void)
{
static char buffer[100] = {0}; // keep it off stack
int x = 0;
int timesTable = 0;

        /*
     * get user to enter the times table, and
     * convert it to an integer for later use
     */

    printf ("Which times table do you want: ");
    gets (buffer);
    timesTable = atoi (buffer);
```

```
    /*
     * output times table via for loop
     */

    for (x = 0; x <= 12; x++)
    {
        printf ("%d x %d = %d\n",
        timesTable, x, timesTable * x);
    }    /* end for */


    printf ("\nDONE!\n");
    return 0;
}    /* end main */
```

In the above program, notice that after we've obtained user input, the **for** statement will run from the values 0 through 12 inclusive, allowing a printout of the appropriate times table to be generated. This is typically how **for** statements are programmed: if you need to run through a specific number of loops, simply code the number of loops as part of the initialization and boolean conditions, and you are guaranteed to execute that many times.

One final word about **while**, **do ... while** and **for** loops: just like their cousin the **if** statement, these loop constructs can be nested inside of each other too. You are capable of having a loop within a loop if the programming problem you are solving demands such a solution.

## 10.4. Other Features of Loops

There are two special **keywords** that C provides that allow us to manipulate loops out of the ordinary manners we seen thus far.

Often, a programmer may wish to abort a loop early, without having to complicate the conditional expression that controls the loop. This is done using the statement **break**, which instructs the C compiler to generate code that will leave the loop if this statement is executed. For example:

```
for (x = 0; x < 1000; x++)
{
    ...
    if (key == 'Q')
    {
        break;
    }/* end if */
}/* end while */
```

The above loop will execute normally as long as the user does not press 'Q' on the keyboard. Once the user has done this, the body of the **if** will execute the **break** statement, causing the loop to terminate. In C, early or abnormal exits from a loop are possible, and usually cases of extraordinary circumstances (such as fatal errors) can be handled with ease to allow loop termination.

The second keyword is **continue**. This is the almost the opposite of **break** in that it causes the loop to move to the next iteration faster than would normally have occurred. Again, an example will make this concept clear:

```
for (x = 0; x < 5; x++)
{
    if (x == 2)
    {
        continue;/* skip value 2 */
    }
    printf ("x: %d\n", x);
}/* end for */
```

The output of this program appears as such:

```
x: 0
x: 1
x: 3
x: 4
```

As the comment suggests, the use of **continue** above will cause the loop to essentially "ignore" the case where **x** is equal to 2. A programmer may write a loop to run through a list of values (as above) and skip the unwanted values, rather than trying to program a complicated set of loops to maneuver through a series of disjointed values.

**Useful Tip: break and continue in if statements**

**break** and **continue** break out of loops. They do not have any effect on if statements.

## 10.5. The switch Statement

We will next examine the flow of control statement **switch**. In this chapter, we introduced the concept of being able to chain **if** statements together, which works quite nicely in being able to detect a specific value in a variable. When a specific value is found, an appropriate piece of code may be executed. This type of situation lends itself quite nicely to the programming of menu type situations, where a menu is presented to the user, and the user must enter their choice. The choice is scanned using a chained series of **if** statements until all choices have been exhausted. If a choice is found, the appropriate code can handle that choice, otherwise an error can be generated for the user.

The **switch** statement allows us to program such sequences without resorting to a long line of **if ... else** statements. The **switch** statement allows us to compare specific cases against the value of a variable the **switch** will operate upon. This flow of control statement allows a programmer to list the cases in a very easy to read and understand manner.

The **switch** statement has the following notation:

```
switch (variable)
{
case CONSTANT1:
```

```
    statement1;
    break;

case CONSTANT2:
    statement2;
    break;

. . .

default:
    statementDefault;
    break;
}    /* end switch */
```

Notice that the **switch** outlines the possible cases a variable is expected to take on, giving a simple, bird's eye view of the possibilities that exist for the different expected values.

The constant that comes after the **case** keyword is a constant that represents a value expected. If the programmer were expecting the value 5, the **case** would read as:

```
case 5:
```

The statements that come after the **case** comparison are executed if the comparison works out true. That is if, say, **CONSTANT1** matches the current value of **variable**, whatever statements come after the **case** will be executed. One statement or 100 statements may come after the **case** line.

The individual cases are terminated using a **break** statement, much like a loop was broken using **break**. Because a case is terminated using the **break**, a compound statement is not needed for multiple lines of code for a particular case.

You can have as many **case** statements as you like to ensure that all possible conditions are checked for. This format is much more compact and readable that a series of chained **if else** style comparisons, although the code generated is almost identical.

You will notice that we have a special case at the end of the **switch**, the **default** case. This is the case that is executed when all others fail, and is a convenient way of saying "if it ain't any of the ones above, then do this one!". In the example of a menu, the **default** case can inform the user they've made an incorrect choice. A programmer may wish to leave out the **default** case if there is nothing special to be done on unexpected values.

**Useful Tip: Location of default**

It is usual but not required to put a default case at the end of a switch. If it would make more sense elsewhere, you can relocate it. An example of when it would make more sense elsewhere is when your default is shared with another case and you have a desired order for cases (e.g. alphabetical and your default has the same behaviour as pressing 'b').

It is interesting to note that if a programmer wanted to do the same code for more than one case of a variable, the programmer simply puts more than one **case** one after another. For example:

```
switch (x)
{
case 1:
case 2:
case 6:
     /* do the code here */
     statement1;
     break;

default:
     /* do default code here */
     statement2;
     break;
}    /* end switch */
```

In the example above, if **x** is equal to either 1, 2 or 6, then **statement1** will be executed. This is a simple method of being able to check for multiple values that are expected to use the same C code.

Another feature of **switch** is that if two or more cases do virtually the same work, except one case has a slightly different action than the other, a programmer can allow cases to "fall through" to the next case in the list. For example:

```
switch (x)
{
case 0:
     y = 5;
     /* fall through to case 1 */
case 1:
     z = 10;
     break;

case 2:
     z = 20;
     break;
}    /* end switch */
```

Notice that in case 0, we set variable **y** to 5, but there is no **break** statement. Because of the missing **break**, we automatically "fall through" to the next case (without any comparisons!), where variable **z** is set to 10. We then hit a break statement, causing the

**switch** to terminate. Thus, the only difference between cases 0 and 1 is that case 0 happens to set **y** to 5. If you find that your programming cases are identical except for a few lines of code, the above approach may be beneficial.

**Useful Tip: Switch Fall-through Situations**

You should put a comment whenever you have a fall-through situation, so that anyone looking at your code knows that you meant to leave the **break** out. It is a good idea to explicitly mention the case that you are falling through to in case you (or some nepharious wrongdoer) decides to put another case after this one instead.

That all being said, there are limitations to what you can compare using a **switch** statement. Some general rules are that the items must be able to be compared using the equality operator, that your case values must be constants, and that you cannot compare floating point numbers.

**Useful Tip: Switch Indentation**

There is quite a bit of controversy over how to properly indent a switch statement. There are arguments for having the case statements further indented from the switch statement or not. There are also arguments for indenting the code that comes after the case statement beyond the standard single indentation from the switch. The best thing to do is to choose a style and stick with it.

## *10.6. The Evil goto Statement*

There is a **goto** statement in C. The language is set up so that you should never have to use one, though. If you really, really, really feel that you have no way around it, examine the problem carefully and see if it couldn't be done using the other control flow constructs (e.g. if, while, do, for, switch, break, continue) in conjunction with a variable to allow you to get out of where you don't want your code to be. This is most common when trying to break out of nested loops or loops that contain switch statements.

**Course Requirement: goto**

Never use a goto statement. If you do, you will lose TONS of marks.

**Useful Tip: goto**

The use of goto may also cause the end of the world as we know it.

## *10.7. Common Beginner Errors*

- Don't forget the semicolon at the end of the do-while.
- Don't forget that a for statement must have exactly two semicolons within the round brackets.

- When you create a switch statement, make sure that the break statements are everywhere where they should be. An easy way to do this is to instinctively type the break statement immediately after you type the line that contains the case statement.

## *10.8. Summary*

The use of the do - while, for, break, continue, and switch statements was introduced. The use of the goto statement was introduced in order to make you not use it.

# Chapter 11. Standard Library Functions

Previously, we learned the intricate details of how to write our own functions. Now we will explore some of the most widely used **library** functions that come with virtually every C compiler, on virtually any operating system platform.

A library function is a function that has been previously written by another programmer, and has been added to a file called a **library file**, for the sole purpose of being included with the code a programmer may be writing.

The use of library functions by a programmer frees the programmer from "reinventing the wheel", so to speak. Since most of the functions supplied as part of the libraries for C compilers are general input, output, file, math and string handling functions, a programmer can concentrate on solving a programming problem from a higher level, writing less low level code.

Library functions are compiled at the company that produces the compiler, and placed into a library file for linking with the code a programmer will produce. The library file is therefore a collection of object files that contain functions that allow manipulation of input, output, files, math and strings.

As you should recall, the purpose of the linker program is to take the object file output of a compiler and link it in with other object files, as well as whatever library files are necessary to create a complete executable file.

Generally, compiler installation software will copy the most used libraries on your system when setting up your compiler, allowing you to have a good selection of predefined functions at your disposal. We have already seen one of these functions in action: **printf()**. This is the standard library function supplied with all C compilers to get information from inside of your software to the outside world through the computer screen. There are other library function variations on **printf()** that allow output to go to the printer, modem, or even a file. Similar library functions exist to get information from the keyboard into your software. A programmer, unless absolutely necessary for speed or efficiency purposes, will never need to write a line of character based input/output low level code since just about everything needed has been predefined in the standard libraries.

We will now begin a look at the various types of library functions available grouped according to their purposes. This chapter will introduce the various functions using the **function prototype** format as described in the previous chapter. The header files needed will also be given, as well as a brief programming example that uses the named functions. The two groups of functions we will look at here are the input/output oriented functions as well as the math-oriented functions. Other library functions will be introduced in other chapters as the need arises. Please note that there are some data types used in the following discussion that have not yet been introduced, but should not affect the general understanding of the functions being demonstrated. Input Functions

The first group of library functions to study is the input functions.

Name:

**char *gets (char *memory);**

Header:

**stdio.h**

Description:

The **gets()** function is used to obtain a string of characters from the keyboard, and place them into memory in a character buffer (see the chapter on arrays) pointed to by **memory**. Although we have not yet seen what a character buffer/pointer are, as well as the strange use of the * (see the chapter on pointers) in the above prototype, it is a very simple function to use if you follow the example below as a guideline on how to use this function. If successful, it will return **memory**, otherwise it will return **NULL** (defined in **stdio.h**).

```
#include <stdio.h>

int main (void)
{
char buffer[100] = {0};  /* declare input buffer */

     printf ("Enter a string and hit <enter>: ");
     gets (buffer);
     printf ("The string the user has entered is: ");
     printf (buffer); /* yes, you can do this */
     return 0;
}    /* end main */
```

Note: gets()'s biggest downfall is that the function has no knowledge about how big the buffer is in terms of bytes. A much more robust function to use is **fgets()**.

**Useful Tip: Visual C++ Deprecation / Unsafe Warning**

If you are compiling the above code using Visual C++ 2005 or 2008 (or newer), you may get a warning or error saying that the gets() function is deprecated or that it might be unsafe. That means that Microsoft has decided that you shouldn't use gets() any more but that it will keep gets() support in until they decide to take it out. To eliminate this warning for this and other such functions (and you will get it a lot), use the following line after your #include statements:

*#pragma warning(disable: 4996)*

Name:

**fgets(char string[ ], int n, FILE *fp)**

Header:

**stdio.h**

Description:

**fgets()** gets a line from a file, similar to what **gets()** does with the keyboard. However, if you use the special variable **stdin** as the file, you can get input from the keyboard instead. The second parameter gives you length protection since it reads at most n-1 characters (it knows to leave room for the null-termination (explained in the section on strings)). If, however, it reads less than n-1 characters, it leaves the '\n' in the string as an indicator. To get rid of it, use:

```
    if( strlen(buf) != 0)
    {
        buf[strlen(buf)-1] = '\0';
    }
```

Modifying the example from gets() is easy:

```
#include <stdio.h>

int main (void)
{
char buffer[100] = {0};  /* declare input buffer */

    printf ("Enter a string and hit <enter>: ");
    fgets (buffer, 100, stdin);
    printf ("The string the user has entered is: ");
    printf (buffer); /* yes, you can do this */
    return 0;
}    /* end main */
```

Name:

**int getch (void);**
**int getche (void);**

Header:

**conio.h**

Description:

These functions will extract a character directly from the keyboard, meaning no buffering of characters will be done. **getch()** will get a character without echoing that character on the screen, whereas **getche()** will do the same but place the character typed on the screen as well. These functions will wait until a character is typed before returning with the ASCII value of that character.

Two important notes: 1) These are often **not portable** to other operating systems. 2) Visual C++ v6 does not handle getch() and getche() very well when combined with other I/O functions.

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

int main (void)
{
int key = 0;

    printf ("Enter a single character: ");
    key = getch();
    printf ("The key entered is: %c", key);
    return 0;
}    /* end main */
```

Name:

**int atoi (char *string);**
**long atol (char *string);**
**double atof (char *string);**

Header:

**stdlib.h**

Description:

These functions will take the character pointer parameter and convert the string of characters into an integer (by **atoi()**),a long (by **atol()**), or a double (by **atof()**) for use in mathematical computations. This is necessary since math cannot be done on a string of raw characters typed in from the keyboard. The characters must be converted into the appropriate data type before any mathematical operation is performed.

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

int main (void)
{
char buffer[100] = {0};/* buffer of 100 chars */
int x = 0;

    printf ("Type in a number & hit <enter>: ");
    gets (buffer);

    /* convert string into an integer */
    x = atoi (buffer);

    printf ("The number is: %d\n", x);
    printf ("The number times 5 is: %d\n", x * 5);
```

```
    return 0;
}/* end main */
```

**Useful Tip: atof() Has a Misleading Name**

Even though atof() implies that it produces a float, **it doesn't**. It produces a double. If you don't include the appropriate include file so the prototype is available to the compiler, your result will be wrong.

## 11.1. Mathematical Functions

This section will introduce some of the most widely used mathematical functions that the standard libraries of C provide for the convenience of the programmer. Advanced mathematical operations such as square roots have to be programmed in C because there is no such operator in the C language.

Prototypes will be given for most of the commonly used math functions. All math functions require the header file **math.h** to be included with your C source code. Also, all math functions require double precision floating point parameters and will return double precision floating point values.

```
/* trigonometric functions */
double cos (double x);
double sin (double x);
double tan (double x);
double acos (double x);
double asin (double x);
double atan (double x);
/* square root */
double sqrt (double x);
/* logarithms */
double log (double x);
double log10 (double x);
/* base raised to the power of exponent */
double pow (double base, double exponent);
```

An example of a few of these functions:

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

int main (void)
{
double a = 0.0;
double b = 0.0;
double c = 0.0;
double d = 0.0;
```

```
    a = 33.67;
    b = 1024.567564;

    /* take square root of b */

    c = sqrt (b);

    /* raise a to the power 5 */

    d = pow (a, 5.0);

    /* print results */

    printf ("sqrt of %f is %f\n", b, c);
    printf ("%f to the power of 5 is: %f\n", a, d);
    return 0;
}    /* end main */
```

Name:

**int abs (int x);**
**long labs (long x);**
**double fabs (double x);**

Header:

**math.h** (and **stdlib.h** for **abs()** and **labs()**)

Description:

The above functions will change their respective parameters into their absolute values. An absolute value means that if a number is negative, the number will be changed to its positive value, and if it is already positive, nothing happens.

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

int main (void)
{
int x = 0;
int y = 0;

    x = -33;
    y = abs (x);
    printf ("x: %d y: %d\n", x, y);
    return 0;
}/* end main */
```

## 11.2. String Handling

Name:

**size_t strlen (char *s);**

Header:

**string.h**

Description:

This function will compute the length of a null terminated string. It returns it as a **size_t** item, which is typically the same as a primitive data type. Which primitive data type **size_t** is equivalent to depends on the system. Under Visual C++ v6, **size_t** is equivalent to an unsigned long, indicating that strings can be over 4 billion characters long.

Name:

**int strcmp (char *s1, char *s2);**
**int strncmp (char *s2, char *s2, int n);**

Header:

**string.h**

Description:

The first function will compare the ASCII contents of two strings, pointed to by **s1** and **s2**, and return a value based upon the following table:

- < 0 if s1 is less than s2
- 0 if s1 is identical to s2
- >0 if s1 is larger than s2

The second function will compare at most **n** characters before returning a result.

Name:

**char *strcpy (char *s1, char *s2);**
**char *strncpy (char *s1, char *s2, int n);**

Header:

**string.h**

Description:

The first function will copy the bytes pointed to by **s2** to the memory pointed to by **s1**. A programmer must ensure that the memory pointed to by **s1** is dimensioned large enough to hold the bytes pointed to by **s2** and its null terminator.

The second function will copy at most **n** characters before returning. Both functions will return **s1**.

Name:

**char *strcat (char *s1, char *s2);**
**char *strncat (char *s1, char *s2, int n);**

Header:

**string.h**

Description:

The first function will concatenate the characters pointed to by **s2** to the end of the string of characters pointed to by **s1**. See the warning about **s1** from the previous function. The second function will concatenate at most **n** characters before returning. Both functions will return **s1**.

## *11.3. Character Classification Functions*

Name:

**int isalnum (int c);**

Header:

**ctype.h**

Description:

This is a macro that will return TRUE if the ASCII character **c** is between A to Z or a to z or 0 to 9 inclusive, otherwise FALSE will be returned.

Name:

**int isalpha (int c);**

Header:

**ctype.h**

Description:

This is a macro that will return TRUE if the ASCII character **c** is between A to Z or a to z inclusive, otherwise FALSE will be returned.

Name:

**int iscntrl (int c);**

Header:

**ctype.h**

Description:

This is a macro that will return TRUE if the ASCII character **c** is between ASCII 0x00 to ASCII 0x1f inclusive or ASCII 0x7f, otherwise FALSE will be returned.

Name:

**int isdigit (int c);**

Header:

**ctype.h**

Description:

This is a macro that will return TRUE if the ASCII character **c** is between 0 to 9 inclusive, otherwise FALSE will be returned.

Name:

**int islower (int c);**

Header:

**ctype.h**

Description:

This is a macro that will return TRUE if the ASCII character **c** is between a to z inclusive, otherwise FALSE will be returned.

Name:

**int isprint (int c);**

Header:

**ctype.h**

Description:

This is a macro that will return TRUE if the ASCII character **c** is printable character (ie: not a control character), otherwise FALSE will be returned.

Name:

**int ispunct (int c);**

Header:

**ctype.h**

Description:

This is a macro that will return TRUE if the ASCII character **c** is any printable character that is not alphanumeric or a space, otherwise FALSE will be returned.

Name:

**int isspace (int c);**

Header:

**ctype.h**

Description:

This is a macro that will return TRUE if the ASCII character **c** is a space character (' '), otherwise FALSE will be returned.

Name:

**int isupper (int c);**

Header:

**ctype.h**

Description:

This is a macro that will return TRUE if the ASCII character **c** is between A to Z inclusive, otherwise FALSE will be returned.

Name:

**int isxdigit (int c);**

Header:

**ctype.h**

Description:

This is a macro that will return TRUE if the ASCII character **c** is hexadecimal digit (0 to 9 inclusive, a to f inclusive or A to F inclusive), otherwise FALSE will be returned.

Name:

**int tolower (int c);**

Header:

**ctype.h**

Description:

This function will convert **c** into a lower case character, if and only if it was an upper case character.

Name:

**int toupper (int c);**

Header:

**ctype.h**

Description:

This function will convert **c** into an upper case character, if and only if it was a lower case character.

## 11.4. Miscellaneous Functions

Name:

**void exit (int value);**

Header:

**stdlib.h**

Description:

This function allows you to terminate your program at any point in the program. The caller of the program (the operating system, for example) will receive the **value**.

The use of the **exit()** function should be carefully weighed as its use causes significant problems with the destruction of objects in C++.

## 11.5. Common Beginner Errors

- Make sure that the appropriate include files are included for the prototypes.
- Don't forget that fgets() will often have an extraneous '\n' at the end of the string.
- Make sure to use the #pragma mentioned in the previous Useful Tip to eliminate extraneous warnings.

## 11.6. Summary

In this chapter we had an introductory look at some of the various predefined functions most C compiler manufacturers supply for us. These functions are all part of the standard library of functions, and enable a programmer from (hopefully) reinventing the wheel in many cases. The use of these functions is unlimited, and the main requirement is that the header files necessary for the individual library functions be included so that prototype definitions can be compiled along with your code.

# Chapter 12. Simple Pointers

Pointers are one of the most difficult topics in C. We will introduce the concept of a pointer variable in a (hopefully) gradual way in order to lead us to more advanced topics. The following is adapted from *C by Dissection* by Al Kelley and Ira Pohl (Addison Wesley, Toronto, 2001, ISBN 0-201-71374-8, pages 264 to 265, 267, and 271 to 272)

## *12.1. Pointer Syntax*

If v is a variable (such as which we've been used to since early in the course), then **&v** is the location in memory of its stored value. This location is also referred to as its **address.** Pointer variables can be declared in programs and then used to take addresses as values. The declaration

```
int i = 0;
```

followed by

```
int *p;
```

declares **i** to be of type **int** and **p** to be of type "**pointer to int**". The legal range of values for any pointer always includes the special address 0 and a set of positive integers that are interpreted as machine addresses on a particular C system. Typically, the symbolic constant **NULL** is defined as zero in **stdio.h**. Some examples of assignment to the pointer **p** are:

```
p = &i;
p = 0;
p = NULL;
```

In the first example, we think of **p** as "referring to i", or "pointing to i", or "containing the address of i". The second and third examples show assignments of the special value zero to the pointer p. It is considered a special value because nothing can be validly stored there.

## *12.2. Addressing and Dereferencing*

We want to examine some elementary code to show what is happening in memory. Let's start with some declarations:

```
int a = 7;
int *p = NULL;
```

The second declaration is read as "**p** is a **pointer to an int** variable with an initial value of NULL." Anytime one declares a pointer variable, it has an underlying type such as int that it can reference. The declaration of **a** causes the compiler to allocate memory space for an int. The declaration of **p** causes the compiler to allocate memory space for a pointer to int. After the assignment statement:

```
p = &a;
```

we can use the pointer **p** to access the value stored in **a**. This is done through the **dereference** (or **indirection**) operator *****. This operator is unary (as opposed to the

multiplication use of the asterisk). Because **p** is a pointer, the expression ***p** has the value of the variable to which **p** points. The pointer **p** is said to have a value which is the address of the int variable **a**. The term **indirection** is taken from assembly programming. The direct value of **p** is a memory location, whereas ***p** is the **indirect** value of **p**, namely the value at the memory location stored in p. Consider the statement

```
printf("p points to %d\n", *p);
```

Because **p** points to **a** and **a** has value 7, the **dereferenced** value of **p** is 7 and that is what is printed. Now consider

```
*p = 3;
printf("a is %d\n", a);
```

The first statement is read "the item pointed to by p is assigned the value 3." Because **p** points to **a**, the stored value of **a** in memory is **overwritten** with the value 3. Thus, when we print out the value of **a**, 3 is printed. This is an **indirect** way of changing the value of a variable through the use of a pointer variable.

In simple terms, the **dereference** operator ***** is the inverse of the **address** operator **&.** Consider the following code:

```
double x = 1;
double y = 2;
double *p = NULL;
p = &x;
y = *p;
```

First, **p** is assigned the **address of x**. Then **y** is assigned the value of the item **pointed to** by p. The last statement is equivalent to

```
y = *(&x);
```

which is equivalent to

```
y = x;
```

A pointer variable can be initialized in a declaration but the notation is confusing to the beginning programmer. Here is an example:

```
int a = 0;
int *p = &a;
```

This declaration tells the compiler that p is a pointer to int and that the initial value to p is &a. Do not read this as "the item pointed to by p is initialized to the address of a."

**Useful Tip: Initializing a Pointer in a Declaration**

When declaring a pointer variable, you should always realize that the asterisk indicates the data type and that dereferencing is not happening.

## *12.3. Pass-by-reference*

Whenever variables are passed as parameters to a function, their values are copied and any changes within the function are not reflected in the calling environment. This **pass-**

**by-value** mechanism is strictly adhered to in C. An alternative is to use the **addresses** of the variables to achieve the effect of **pass-by-reference** and, thus, be able to change the contents of the variables in the calling environment.

For a function to achieve pass-by-reference, we must tell the function where the variables that we wish to modify are located. To do so, pointers must be used in the parameter list in the function definition. Then, when the function is called, addresses of variables must be passed as parameters.

## 12.4. Pass-by-reference Example

The function swap() in the following program illustrates these ideas.

```
#include <stdio.h>

void swap(int *p, int *q);

int main (void)
{
int a = 3;
int b = 7;

     printf("%d %d\n", a, b);
     swap(&a, &b);
     printf("%d %d\n", a, b);
     return 0;
}    /* end main */
```

Because the addresses of **a** and **b** are passed as parameters to **swap()**, the function is able to exchange the values of **a** and **b** in the calling environment.

```
void swap(int *p, int *q)
{
int tmp = 0;

      tmp = *p; // First line of interest
      *p = *q;  // Second line of interest
      *q = tmp;      // Third line of interest
}    /* end swap */
```

We read the three noted lines above as follows:

- The variable **tmp** is assigned the value of the variable pointed to by **p**.
- The variable pointed to by **p** is assigned the value of the variable pointed to by **q**.
- The variable pointed to by **q** is assigned the value of **tmp**.

After these three statements have been executed, **a** has the value 7 and **b** has the value 3.

## 12.5. Another Example Of Pass-By-Reference

Here's another example of using **pass-by-reference** to return multiple values from a function:

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

void convertHms (double a, double *b, double *c, double *d);

void convertHms (double a, double *b, double *c, double *d)
{
double x = 0.0;

    *b = (int)a;/* cast to an int to chop decimals */
    x = (a - *b) * 60.0;
    *c = (int)x;
    *d = (int)((x - *c) * 60.0);
}    /* end convertHms */

int main (void)
{
double hour = 0.0;
double h = 0.0;
double m = 0.0;
double s = 0.0;
char buffer[100] = {0};

    printf ("Enter hour (ie: 10.25): ");
    gets (buffer);
    hour = atof (buffer);

    convertHms (hour, &h, &m, &s);
    printf ("hour: %f = h: %f m: %f s: %f\n",
        hour, h, m, s);
    return 0;
}    /* end main */
```

The above program asks the user to enter the time in a fractional format (meaning that you enter the time using fractions of an hour) and will convert that fractional hour into the appropriate hour, minute and second. For example, 10.25 hours will be converted to 10 hours, 15 minutes and 0 seconds, and 22.341666667 will be converted into 22 hours, 20 minutes and 30 seconds.

Notice that there are three values that must be returned by a function which will convert the fractional hours. We could have made the **double**s **h**, **m** and **s** global, and had the function **convert_hms()** set the globals with the appropriate values, but that is messy and also bad programming style. A much more elegant solution is to pass to **convert_hms()** pointers to the three variables it must fill: a pointer to **h**, a pointer to **m** and a pointer to **s**.

The ampersand tells the C compiler to generate code that calculates the address in memory where the variable exists. The address-of operation always generates a pointer to the variable it operates on. To give **convert_hms()** pointers to **h**, **m**, and **s**, we use the address-of operator: **&h**, **&m**, and **&s**. Note that this generates three double precision pointers (**double ***).

The actual function **convertHms()** has been declared to accept a **double** parameter **a**, and 3 **double pointer** parameters. Parameters **b**, **c**, and **d** are all pointer variables to double precision floating point numbers in memory. The function will proceed to convert the number passed in **a** into the hours, the minutes and the seconds. Notice that the converted values are supplied to the caller of this function using the ***** (contents-of) operator, which is the opposite of the address-of operator. The contents of, say, parameter variable **c** in this function are filled with the converted minutes, therefore, in this manner, the local in **main()** called **m** will be able to obtain the minutes from **convertHms()**.

This technique is extremely valuable in terms of creating functions that are meant to supply back more than one piece of information. Because the return value mechanism is limited to one piece of information at one time, passing in addresses of variables to fill in is a wonderful solution, and ensures that functions can be expanded infinitely.

NOTE: A better name for **convertHms()** would be **convertHourMinuteSecond()** to avoid the issue of "what's an Hms?" :-).

## 12.6. Casting

You will notice a curious expression used in the **convertHms()** function:

```
*b = (int)a;
```

The keyword **int** inside the round brackets is used to convert, or **cast** (alter) the data type of variable **a** temporarily from a double precision floating point number to an integer. Using an **int** cast against a double precision floating point value will truncate, or chop off, the fractional digits, leaving only the whole number. This is how we can obtain the integer portion of our fractional hour. 10.25 will be converted into the integer value 10, and because it is to be stored in the contents of an double precision memory location pointed to by **b**, it is automatically casted or reconverted back into a **double** before being stored inside the memory pointed to by **b**. Casting is used throughout C programs to easily convert from one data type to another.

As a further example of casting, suppose you wanted to take a look at the individual bytes of a short integer (remember that an **short** is two **chars** side by side), try this:

```
short x = 0;
char *a = NULL;
. . .
x = 1234;

/*
 * taking the address of x would
 * normally get an short pointer,
 * so cast to a char pointer!
```

```
 */

a = (char *)&x;
printf ("1st byte (in hex): %x\n", *a);
a++;
printf ("second byte (in hex): %x\n", *a);
```

As the comment above states, using **&x** in your program, (if **x** was an short), would generate an integer pointer (**short ***). The casting expression **(char *)** is used to change the pointer from being an integer pointer to that of a character pointer. Pointers can be created and manipulated for any data type you want, at your free will! By not using casting, however, your program may not compile cleanly as most C compilers will generate errors or warning about pointer type mismatches.

## 12.7. Common Beginner Errors

- Always initialize pointer variables when declared.
- Distinguish between the * and & operators when used in front of variables.

## 12.8. Summary

This chapter briefly introduces pointer variables through the concept of assigning the addresses of variables to pointer variables in order to indirectly change or access values. Pass-by-reference is introduced as a way of returning more than one item from a function. Casting is used to convert between data types.

# Chapter 13. Arrays

## 13.1. Introduction

Over the last few chapters we have introduced enough programming concepts for the C language to enable a programmer to write some rather powerful and interesting software. In this chapter, we will expand this knowledge base by introducing our first complex data object: the **array**. This data object is sometimes referred to as a **data structure** because there is a structure or organization to how the data for this object lies in memory. You will also sometimes see an array referred to as an **aggregate data type**.

In real life, there are many examples of arrays. An accounting worksheet is grouped as a two-dimensional array or matrix of cells (some number of rows by some number of columns) where an accountant may place financial figures, and computations are done on these figures to arrive at subtotals for each column and row. A street of houses is another example. Each house has an **offset** (the house number) from the start of the street, and houses are generally a linear progression from one end of a street to another. Hotel rooms are another real life example.

An array in C can be thought of as a linear sequence of memory locations reserved for the storage of objects of a particular data type. The programmer has the ability to manipulate these linearly arranged objects with the notations introduced in this chapter.

An array in C programming has many uses. An array can be used to hold data obtained from a user, or it can hold characters typed in on the keyboard (for example, the keystroke buffer that is used in conjunction with **gets()**) or even hold memory locations of functions that can be called without using the function name. The possibilities in C are almost endless!

The following is a pictorial representation of an array of short integers, of size 5 elements, starting at address 536 (arbitrary value):

Table 13-1: Array Layout

| **Memory location** | 536 | 538 | 540 | 542 | 544 |
|---|---|---|---|---|---|
| Array offset | 0 | 1 | 2 | 3 | 4 |
| Value | -5 | 123 | 0 | 3192 | -32 |

## 13.2. Base Address

The above table shows that there are five short integers placed in memory, starting at memory location 536. This location is termed the **base address** of the array (or sometimes the **base location**). Notice that the memory locations increment by a value of 2 for each integer: recall that an **short** is composed of two bytes of memory.

## 13.3. Array Offset

The table above also shows the **array offset** to each element in the array. The first element, which is the value **-5**, is at array offset 0 from the start of the array. The next integer is at an array offset of 1.

An array offset is simply the element within the array that we wish to work with, and it corresponds to a location in memory starting at the base memory location, plus the size of the data object multiplied by the offset. This general rule of thumb applies to arrays of all data types: **char**s, **int**s, **long**s, **float**s and of course **double**s. Note, however, that the compiler is responsible for determining where an array object lies in memory, not the programmer. The programmer only needs to use the array offset to reference a particular element in the array.

In the sample array above, the first element is at memory location 536 plus an array offset of 0, which is of course memory location 536. The second element is at array offset of 1. Since a short integer takes two bytes of memory, the actual byte offset is 1 * 2, as a **short** takes two bytes, giving us a memory location of 538. Offset 2 is at memory location 540 (location 536 + 2 * 2). If the sample array were an array of **double**, we would be multiplying the array offset by the size of a double to compute where in memory an element resides. A programmer need not worry about where in memory these objects lie. This discussion simply aids in our understanding of C and a program's memory layout to know how a compiler treats objects in an array.

## 13.4. Array Offset Operator

The generalized notation for accessing an element within an array is simply the name of the array, with the offset inside of the square brackets. Using this notation, a programmer can extract information at that array offset, or place new information into that array offset. In fact, the square brackets **[ ]** are nothing more than a special **operator** that informs the compiler to add an offset, which is scaled by the size of the data type, to a base address.

## 13.5. Notation

In C, when we declare an array variable to the compiler, we must always specify the **dimension** of an array (in some way or another). This is necessary so that the compiler can reserve the correct amount of bytes for the purposes of our program.

The following is an example of a C array declaration:

```
short x[5];
```

This declaration will reserve memory for 5 consecutive short integers in an array called **x** for this example. It should be plain that an array in C is an easy method of being able to reserve large blocks of memory for data storage, and array notation allows simple access to each element in this block.

The generalized notation for declaring a variable to be an array is:

*data-type variable-name[# of elements];*

Once we have our array declared, in our code, we can access an object within an array by using **subscript notation**. This is done by placing the object's array offset inside of square brackets following the array name in our code. Although this is similar to the way we declare an array, we are performing a totally different operation when this is done in code. As an example, using our integer array **x** from above, we can access elements in the array using notation such as:

```
x[2] = 5;        /* assign 5 into object number 2 */
```

C always references array elements starting at the "zeroth" element. In other words, the first element in our array is always **x[0]**, the second is **x[1]**, the third is **x[2]**, and so on. The idea of a zero based array is explained using our array offset discussion: the first element of an array is at array offset zero from the start of the array. The second element is array offset by 1 in the array, and so on.

To help rationalize our understanding of arrays thus far, consider the following scenario. Imagine a program that must keep track of the mileage driven by a sales person for a week, and then report the total mileage. A first crack at this program would probably set up 5 integers, similar to the following:

```
int day1 = 0;
int day2 = 0;
int day3 = 0;
int day4 = 0;
int day5 = 0;
int total = 0;  /* used to hold the total mileage */
```

The 5 integer variables would thus be able to hold all 5 days worth of mileage data.

To generate a report that outputs the total mileage driven by the sales person, we may compute the total as thus:

```
total = day1 + day2 + day3 + day4 + day5;
```

So far, so good, and in fact, this is a trivial programming scenario for anyone thus far.

Now, to complicate matters, the requirement for the program will switch to maintaining the mileage for an entire *month* of time. Now we have some problems. Using the method shown thus far, we would need to create up to 31 *independent* integer variables to hold the mileage data for all 31 days of a given month. This is now becoming cumbersome, as to compute the total mileage for the month, we would have to write code to sum up 31 independent integer variables. It gets worse – imagine keeping track of one full *year* of data!

This is where arrays come into play. Let's return to our original scenario of one business week of data. The following snippet will show how to declare an array to hold the mileage data, and how to sum up the array elements.

```
int mileage[5];
int x = 0;
int total = 0;

. . .

for (x = 0; x < 5; x++)
{
    total = total + mileage[x];
}    /* end for */
```

This is *far* easier to deal with than the 5 independent variables. Note how we declare the array of integers – we have told the compiler to reserve 5 consecutive int data objects in memory. We have our summation variable called total set to zero as before. We have one extra variable, x, which will be used to run a loop.

Recall that an array starts at offset zero, and if the array has 5 elements, the last element of the array will be element number 4. Hence, as long as x is starting at zero, and remains less than 5, we are accessing valid elements of the array.

To sum up the array, we add each individual element of the array to the summation total. This is exactly the same as what we had previously, where we added day1 to day2 to day3, etc.

By running the loop, we now have an *algorithmic* method to generate the total. An algorithm is a software formula to compute a result. Algorithms are important because they will simplify work, in the long term. Here, it appears we have done more work to compute the result of 5 days of mileage. But now consider the task of doing one month. We have a *very* easy way to handle this – change the 5 in the code snippet above to 31, and we're home free!

And to do a year, change the 5 to 365, and instantly, we have the ability to compute one year's worth of data.

Hopefully this illustrates clearly the benefits of a common grouping of data elements under a single name. Arrays are so important to C that this chapter will drill home a number of techniques you can use to implement arrays as part of your programming efforts.

**Useful Tip: Use constants for array sizes**

It is best to use constants (through #define) to define array sizes.

## 13.6. More About Array Offsets

The C language is somewhat strict in how you write software, as you must always define variables, functions, etc. before you get a program to compile correctly. Unfortunately, the compiler is only capable of enforcing syntax rules, and does no sanity checks on whether the code we have written will perform a task adequately.

One of the main dangers of working with arrays is to try to access an element outside of the dimension we have specified. Take for example:

```
long val[10];
```

The above declaration reserves ten long integers in memory one after another, a total of 40 bytes of memory being reserved. In our code, we have access to ten **long**s through the array offsets **val[0]** to **val[9]** inclusive.

If you inadvertently use an array offset of **val[10]**, you should be aware that you are now trying to access the *11th* long integer in the array. However, our array has been dimensioned to hold only 10 long integers. What does the compiler do?

In C, **nothing special**! The compiler will happily generate code to access this 11th **long**. If our code contained something similar to:

```
val[10] = 55L;
```

we have now overwritten a piece of memory past the end of the array with four bytes organized as the value 55. If we had a crucial variable stored after the end of the array, we have now destroyed the contents of that variable, which will lead to all kinds of havoc in our program. For this reason, C programmers must be extra vigilant in keeping array offsets within their bounds.

It is up to the programmer to ensure that what an array offset is doing is legal for the particular application. Array offsets such as:

```
x[-2] = 5;
```

are legal. It is only valid, however, as long as what is in memory two integers **before** the start of the array is something that can be overwritten without worry. (The example above is something that is done more often with pointers than with arrays.)

## 13.7. Initialization of Arrays

Often in C, we would like to set up an array so that the array values we need are ready to go when we execute our code. There are three ways to accomplish this. Study the first method below:

```
int main (void)
{
```

```
int anArray[7];            /* declaration */

    anArray[0] = 100;
    anArray[1] = 32700;
    anArray[2] = -31231;
    anArray[3] = 0;
    anArray[4] = 256;
    anArray[5] = 255;
    anArray[6] = 4321;
    . . .
}    /* end main */
```

The above code declares an array of 7 integers, and later in our code, we give individual elements of the array their values. However, to initialize a number of elements in this fashion is awkward in terms the amount of repetitive code is required.

Recall how to initialize a simple variable:

```
int x = 5;
double y = 10.33;
```

For array initialization we use the following second type of initialization:

```
int anArray[7] =
{
    100, 32700, -31231, 0, 256, 255, 4321
};
```

Here we have an initialization of the array **anArray**, and is useful to supply your C program with values at compile time, rather than at run time.

The third method will seem to contradict an earlier statement:

```
int anArray[] =
{
    100, 32700, -31231, 0, 256, 255, 4321
};
```

The array appears to be dimensionless, due to the use of **[]** (the empty square brackets). The compiler will not complain about this declaration, however, as the programmer is supplying 7 integers as initial values for this array. The programmer has effectively told the compiler that the array has an **implicit** dimension of 7. This notation is useful in many circumstances where the contents of an array are being modified in the source code often, so rather than constantly change the dimension, leave it out, and let the compiler figure out the size of the array from the number of initialization values.

In the latter two examples, what the C compiler will do is allocate a block of memory for the array, and fill in the values for you, saving the trouble of writing the code to do the assignments one element at a time.

What follows is a more concrete example of where using and initializing an array can simplify a program tremendously:

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

```
int calcNumDays (int m, int d);

int daysPerMonth[12] = {31,28,31,30,31,30,31,31,30,31,30,31};

int calcNumDays (int m, int d)
{
int numDays = 0;
int x = 0;

    /* add from jan to last fully completed month */
    for (x = 0; x < (m - 1); x++)
        numDays += daysPerMonth[x];
    numDays += d;
    return (numDays);
}   /* end calcNumDays */

int main (void)
{
char buffer[100] = {0};
int month = 0;
int day = 0;
int total = 0;

    while (1)
    {
        printf ("Enter month (999 to end): ");
        gets (buffer);
        month = atoi (buffer);
        if (month == 999)
        {
            break;
        }


        printf ("Enter day: ");
        gets (buffer);
        day = atoi (buffer);

        total = calcNumDays (month, day);
        printf ("month: %d day: %d total: %d\n",
            month, day, total);
    }/* end while */


    printf ("done!\n");
    return 0;
}    /* end main */
```

As you can see from the math inside the function **calcNumDays()**, the array version of this calculation is much simpler to code and understand than the version we had a few chapters ago using a chained series of **if**s. We have replaced the series of **if** statements with a simple run through an array of values, from the start of the year to the month just *completed*, to get the total days in the complete months elapsed so far. We then add the number of days into the current month, and bingo, we have the total. Of course, error checking should still be done to ensure all results of the program are valid.

**Useful Tip: Initializing Arrays**

If you have **any** elements of an array initialized, any element that you did not explicitly specify are implicitly initialized to 0.

**Useful Tip: When a Table is Better Than Chained if Statements**

It is always much more efficient to code tabular forms of information into an array to be referenced through some form of offset for the particular piece of information that is needed, rather than perform a complicated series of **if** statements to isolate the values desired. NEWDATE.C is a prime example of an array based table of data.

**Useful Tip: while(1)**

**while(1)** means "loop forever". Of course, you can always break out, like we did here.

**Course Requirement: Always Initialize Arrays**

Always initialize your array variables, even if it's just with a single 0 in the very first element.

## 13.8. Multi-Dimensional Arrays

C allows you to have multidimensional arrays. This is a further extension of the idea of an array, which allows you to efficiently organize and access **groups** of information. One of the most common multidimensional arrangements is a **row-column** array, otherwise known as a two dimensional array. Many useful programs use this format - for example, the spreadsheet programs you may use on a day to day basis use a two dimensional array to organize the rows and columns of cells!

With more than two dimensions, an array may get hard to visualize, but to simplify your thought patterns, think of a multidimensional array as nothing more than an array of arrays!

## 13.9. Common Beginner Errors

- Make sure that you don't access past the end of the array.
- Don't confuse the dimensions of multi-dimensional arrays.

## 13.10. Summary

This chapter has introduced the concept of the **array**, which was defined as a linear progression of data objects in memory. The advantage of such a structure is that we can

easily access individual members of such an array using the name of the array and the offset to the data object. It was noted that C arrays are zero based, meaning that the first element is actually at offset zero, the second element at offset 1, and so on. We also saw that in our generated code, offsets are multiplied by the size of the data type in bytes to obtain the byte offset from the base address of the array.

# Chapter 14. Strings

So far, we've glossed over the concept of character strings. Unfortunately, dealing with strings is one of the more awkward parts of the language. Let's address it with an example to lead off.

## *14.1. C and Character Strings*

Let us have a look at a very simple C program which will calculate the length of a string of characters:

```
#include <stdio.h>

/* prototype */
int length (char string[]);

/*
 * note that we are passing an array reference to
 * the function length()
 */

int length (char string[])
{
int len = 0;


    len = 0;
    while (string[len] != '\0') /* '\0' ends string */
        len++;/* increment len */
    return (len);      /* and return it */
}    /* end length */

int main (void)
{
char buffer[100] = {0};

    printf ("Enter a string: ");
    gets (buffer);
    printf ("Length of %s is: %d\n", buffer,
        length (buffer));
    printf ("Length of %s is: %d\n", "hello",
        length ("hello"));
    return 0;
}/* end main */
```

In the **main()** function, we declared a local variable called **buffer** as an array of characters, 100 in total, which will contain the string of characters we type in on the keyboard when we call the **gets()** function. We have just instructed the compiler to reserve, in memory, space for 100 consecutive characters of information.

Near the start of program, the first function we declare is called **length()**. This function is used to calculate the length of any arbitrary string of characters. The term "string of characters" is a fancy term for an arbitrary concept in C programming. In C, a string is defined an array of ASCII (char) values one after another, ending in a **null terminator**, which is the ASCII code with value zero. This is known as **null-termination.**

The null terminator is represented by the escape sequence **'\0'**. This is not a two-character sequence. That is, this is not a backslash followed by a character '0'. This is a way of denoting a single character. The presence of a backslash followed by a number or series of numbers within **single** quotes allow you to specify unprintable characters. The series of numbers indicate the **octal** (see the Appendix on Counting Systems) value of the ASCII character.

A string in C **always** has a null terminator, since this is the programming convention used to inform the various string handling functions in the standard library that the string of characters is terminated (ended) inside memory. As an example, the string "Ab2%" is represented as the following array:

| **Array offset** | **0** | **1** | **2** | **3** | **4** |
|---|---|---|---|---|---|
| **char** | 'A' | 'b' | '2' | '%' | null-terminator |
| **ASCII** | 65 | 98 | 50 | 37 | 0 |

In our example program here, a string is passed to the **length()** function by means of a rather odd looking parameter declaration:

```
char string[]
```

This parameter declaration, with no dimension value inside of the square brackets, informs the compiler that this function is going to be passed a reference (or base address) to some *arbitrary* sized array of characters (a string). However, at the time we develop this function, we are not sure of the exact length. C allows array parameters to be defined in this manner, which offers flexibility in working with arrays, as array-handling functions need not know the exact size of the array it will work with. Often, a second parameter will be supplied that would indicate the maximum number of elements to work with.

The function sets a counter called **len** to zero, and will use this counter to step from element to element inside of the array of characters. As long as the current string array element **string[len]** is not equal to the null terminator, we know we haven't yet reached the end of a C string. (Remember that C strings are zero terminated, meaning that there is **always** a '\0' character at the end of any valid C string!)

As soon as **string[len]** is '\0', we terminate the **while** loop, whose only job was to increment **len**, and then return the current value of **len**. This is the length of the string in characters.

In our **main()** function, we get some user input, and then call the **length()** function to calculate the length of that user input, and finally, display how long the string was (in characters typed). Note how we can pass to the **length()** function a reference to the base address of the keystroke buffer by simply using the name of the array **buffer**. This is how we can pass array references to any function that requires knowledge of the start of an array. The array name all by itself is always the base address of that array.

**Useful Tip: Array Names**

The name of an array is the same as the address of its very first (or zeroth) element. Memorize this.

In **main()** we also print how long the string **"hello"** is. Remember that a string of characters is nothing more than an array of characters with a null-termination, hence we can give any function that needs an array reference a string such as the above. **"hello"** will of course result in a string of length 5.

## *14.2. The Innards of Strings*

We have discussed strings without really describing what the compiler does with them. As we know, a string is normally defined as a series of characters in between double quotation marks such as:

```
"ABCdefGHI"
```

Strings such as the above are used all over in C. For instance, all the formatting information for **printf()** is supplied through the use of formatting strings. Text we want placed on the screen through the use of **printf()** is supplied using strings.

When the compiler comes across a string as shown above, it does two things. Firstly, it will take the characters in the string, and place them aside as it generates the executable file for the source code. As it places them aside, it will also append a **null termination** (the character '\0') at the end of the string in order for the string to be properly defined. The null termination is a marker that is used to tell functions such as **printf()** where the string ends in memory so that these functions don't attempt to print out the character representation of all of memory.

What the compiler puts in the place of the string inside the code is the **base address** of the first character in this string of characters. What **printf()** obtains is the address of where the first byte of the string exists in memory. All that these functions (and any function that expects a character-based string) do is simply run through the characters in the null terminated array one at a time until the null terminator is reached. As far as the C compiler is concerned, a string is an address of where the first character of the string resides in memory, and passing a string as a parameter to a function is simply passing the base address of this array.

If you recall the function **gets()**, it requires a character buffer to be passed as its parameter. As we've seen in this chapter, a character buffer is just an array of characters of some dimensioned size. The character buffer name is the base address of where the first character of the buffer resides. All that **gets()** does is obtain a keystroke from the keyboard, and place that keystroke in the next available location in the character array. When <enter> is hit, **gets()** will place a null termination after the last character received, terminating the string properly.

As you should hopefully see, both quoted strings and character input buffers are one and the same as far as the C compiler is concerned: they are nothing more than base addresses of where their respective first characters lie in memory! This is a crucial concept to understand for the next chapter, which deals with **pointers** - a data type which allows a programmer to point to any piece of data in memory, including arrays of information.

**Useful Tip: Overwriting Null-termination in Strings**

Whenever manipulating the characters within a character array, especially if you are attempting to change the length of a string of characters, it is a good idea to ensure that the null termination is always updated to the correct location inside the array. Accidental overwrites of the null termination with a non-null value is a good (bad?!?) way to crash an application!

## *14.3. String Library Functions*

At the start of the chapter, it was mentioned that dealing with strings in C was awkward. That is because you can't deal with strings with many of the usual operators like =, = =, +, <, >, etc. Instead of using the operators, there exist **string library functions** that are used.

In the next few sections, assume that you have the following two string variables:

```
char str1[30] = "fred";
char str2[30] = "barney";
```

### Assignment using strcpy instead of =

You can't do this:

```
str1 = str2;
```

Instead do this:

```
strcpy(str1, str2);
```

**strcpy()** will **copy** the contents of the second parameter into the first parameter. Thus, the contents of str1 will no longer contain "fred". It will contain "barney". str2 will be unchanged.

There are other variations on strcpy() such as strncpy() that do assignment in slightly different ways as well.

## Comparison using strcmp instead of ==, <, and >

You can't do this:

```
if( str1 == str2 )
```

Instead do this:

```
if( strcmp(str1, str2) == 0 )
```

**strcmp()** will **compare** the contents of the second parameter with the contents of the first parameter. If they are equal, 0 is returned. If str1 is less than str2, an integer less than 0 is returned. If str1 is greater than str2, an integer greater than 0 is returned.

It is important to **explicitly compare** the result returned from strcmp() to 0. Since 0 is equivalent to FALSE, doing

```
if( strcmp(str1, str2) )
```

will give you exactly the **wrong** result if you think it will be true if the two strings are equal.

Comparison of strings is done according to the ASCII values of the characters in the strings. Character-by-character comparison is done until an inequality is found between characters at the same offset in both strings.

There are other variations on strcmp() such as stricmp(), strcmpi(), strncmp(), etc. that do comparison in slightly different ways (such as case-insensitivity) as well.

## Concatenation using strcat instead of +

You can't do this:

```
str1 = str1 + str2;
```

Instead do this:

```
strcat(str1, str2);
```

**strcat()** will **copy** the contents of the second parameter to the end of the contents of the first parameter. The null-termination will be overwritten and put at the end of the new, longer string.

You must make sure that there is enough room in str1 to hold both str1, str2, and a null-termination. The compiler will **not** do it for you!

## strlen to find the length of a string

It is very valuable to know how long a string is. strlen() is a library function that will give you this information. Pass the string as an argument. strlen() returns the length as a **size_t** item, which is typically the same as a primitive data type. Which primitive data type **size_t** is equivalent to depends on the system. Under Visual C++ v6, **size_t** is equivalent to an unsigned long, indicating that strings can be over 4 billion characters long.

## 14.4. Common Beginner Errors

- Don't use normal operators like =, = =, +, etc. to work with strings. Use the string library functions.
- When using strcpy() and strcat(), make sure that you don't access past the end of the array.
- When using strcmp(), explicitly compare the result to 0.

## 14.5. Summary

This chapter introduces the way that C handles working with strings. Selected string library functions were used to replace operators that are not supported by the language.

# Chapter 15. More Pointers

## 15.1. Review of Array Concepts

In the last chapter, we introduced the concept of arrays. The array, as we discovered, is a structured data type that allows access to a linear set of data objects in memory. The array allowed efficient access to objects in memory since the C compiler could generate code that took advantage of this linear nature.

It was stressed that, when working with arrays, the C compiler is mainly concerned with offsets from the **base address** of the array. The base address was defined as the memory location where the array began. When the compiler generates code, the offsets that are added to the base address are scaled by the data object size. An offset for an array of **long**s will be multiplied first by 4 (4 bytes in a **long**), before adding it to the array base address to locate the first byte of that data object.

To refresh our memories about the arrangement in memory for an array, study again the following chart showing a fictitious array of short integers:

Table 15-1: Array Contents

| **Memory location** | 536 | 538 | 540 | 542 | 544 |
|---|---|---|---|---|---|
| Array offset | 0 | 1 | 2 | 3 | 4 |
| Value | -5 | 123 | 0 | 3192 | -32 |

## 15.2. Addresses

Notice that the starting location of our array is 536. This location is an **address** in memory, which is nothing more than a place in memory where data can be stored. It turns out that we can access every single byte of memory in our computer by using the appropriate address.

Addresses for microcomputers typically begin at location 0, and end up at one less than the number of bytes the computer is designed to hold. Although it is useful to understand exactly how your PC accesses memory (both read only ROM and writable RAM), all a program really needs to know is that all bytes of memory (both RAM and ROM) can be accessed using the appropriate addresses for your PC if permitted by your operating system.

The idea behind being able to address a particular location in memory is akin to being able to locate an individual living in a city. We look up the address of that individual, go to that address, and by being let in through the front door, we can see what is inside of that address (for example, the number of occupants). To help you visualize how memory addresses work, think of your computer's memory as being a large city of millions of inhabitants. Each citizen (byte) must have their own address of where they live (there is no such thing as a homeless byte in computerland!). To see who's at home at a particular address, use the memory address of that byte, and have a look at that location (use a **read operation** on that memory location). To move a new person into that address (to change the value), use a **write operation** to change the contents of that address.

As should be obvious from the above analogy, an address in a computer's memory space allows access to the **contents** of a particular memory location so that software can extract those contents, or place new information into that location. This idea of looking at or placing new information into a location in memory is usually referred to as **peeking** and **poking** information into memory.

As another example of an address, let us have a look at a character string in memory:

*“HELLO”*

Table 15-2: Character String Representation In Memory

| **Memory Address** | **1016** | **1017** | **1018** | **1019** | **1020** | **1021** |
|---|---|---|---|---|---|---|
| **Character** | 'H' | 'E' | 'L' | 'L' | 'O' | null-terminator |
| **ASCII value** | 72 | 69 | 76 | 76 | 79 | 0 |

The table above should re-iterate the concept that an address has no particular or specific value: it is simply a location inside the computer's memory where a value can be read or written to. When you write a program in C that uses character strings, arrays, etc., these data objects reside in memory locations that vary from computer to computer, program to program, somewhere inside those memory locations. Addresses allow you to obtain and manipulate those bytes of memory, giving you almost complete control over where you place objects in memory, or what you do with them.

Of course, this power can be (and often is) abused, or even worse, accidentally misused, resulting in mysterious side effects like system crashes, weird characters on the screen and the like.

## *15.3. Initializing a Pointer Variable*

We can declare a pointer variable to access characters of memory:

```
char *stringPointer = "HELLO12";
```

The first line is the declaration of a pointer variable. As you should recall, the declaration begins with a data type, in this case **char**, since a character pointer is simply a pointer to objects of type **char**. Also recall that the asterisk in front of the name of the variable tells the C compiler that we are declaring a pointer to that data type.

To make the pointer useful, we **must make it point to something** in memory. A convenient object to point to is the first byte of a string of characters. We can assign into **stringPointer** the address of where the capital **H** is in the string **HELLO12**. It appears that we are setting our pointer variable equal to the string, but, if you remember from our discussion about what the compiler does with string constants, all the double quotes of the string indicate is that an address should be generated. This address will indicate where the first character of the string lies in memory. The assignment of the string **"HELLO12"** to the pointer variable is really the assigning of the address of where in memory the first byte of those 7 bytes (plus null terminator) exist.

If the string **"HELLO12"** happened to reside in memory locations 1016 through to 1023, **stringPointer** will get the value 1016. The reason for this is because of the fact **stringPointer** is a pointer variable: the string begins at that address, therefore the pointer variable will get that address. It cannot be stressed enough how important this concept is for a good understanding of how pointers work.

We will look at other initialization techniques later in this chapter, for other pointer types.

At this point, we should introduce a constant that is heavily used with pointers. **NULL** represents an address (typically address 0) that cannot be accessed. It is customary to initialize pointer variables to NULL if the variable isn't pointing to something valid. NULL is usually defined in stdio.h or stdlib.h.

**Course Requirement: Pointer Initialization**

Always initialize pointer variables. If you don't know what a valid value would be for the pointer when you declare it, initialize it to NULL.

## *15.4. Accessing Memory Using Pointers*

When using a pointer in C, the programmer has the ability of looking at memory (peeking at the contents of an address), or storing new information in a memory location (poking information into that address).

Now that we have an initialized character pointer we can do a number of things. In the chapter on arrays, we studied the function called **length()** which could be used to compute the length of a character string. (This function, by the way, is always included in the standard libraries of every C compiler as a function called **strlen()**). With our string pointer variable, we can pass to that function the address of our string in memory. Recall that an array simply references the first memory location where our consecutive set of data objects resides. Our variable **stringPointer** is doing this exact same operation! In fact, you can view an array as a **fixed address pointer**: the address cannot change lest we lose our reference to our reserved memory.

The way we work with arrays is the same way we can work with pointers. Let's have a look at two examples:

```
char x = 0;
. . .
x = stringPointer[0];      /* array notation */
x = *stringPointer;        /* pointer notation */
```

The first line above appears to be an array reference: indeed, since a pointer variable can be used just like an array, you can think of **stringPointer** as being an array of chars. It has an added benefit: **stringPointer** can be changed to point to **any** string of characters. Using an array offset of 0 from where **stringPointer** is pointing to, we see that **x** will get the value 72 (the letter 'H').

## 15.5. 'Contents of' Operator

The second example above has some new syntax. The statement **x = *stringPointer** is read as "**x** is assigned the **contents of** the address referenced by **stringPointer**". Since **stringPointer** is currently set to address 1016, the variable **x** gets what's in memory at that address, namely 72. This uses a principle called **dereferencing the pointer** (using an asterisk in front of the pointer variable)**.**

Now you may be wondering: "if they're both the same, why bother using the weirder looking second example"? The answer lies in the fact that using pointers is almost always more efficient in terms of the amount of executable code produced by the C compiler. Most microprocessors are wonderfully proficient when working with addresses, but are not as effective at working with offsets from the start of an array. The first line above would cause the following code to be generated:

```
a) obtain the address contained in stringPointer
b) obtain the offset, and scale by the data object size
c) add scaled offset to that address
d) get the contents of that address
```

whereas the second example generates:

```
a) obtain the address contained in stringPointer
b) get the contents of that address
```

The middle steps of scaling and adding the offset are not done using pointers since the pointer is *already where we want it*.

Pointer variables are not just limited at looking at a single memory location. We can apply any mathematical operation to a pointer variable to move the pointer (the address that the pointer variable holds) anywhere we want, although for all intents and purposes, only addition and subtraction are really useful. Again, an example:

```
stringPointer++;      /* example 1 */
x = *stringPointer;        /* example 2 */
stringPointer += 3;        /* example 3 */
x = *stringPointer;        /* example 4 */
x = *--stringPointer; /* example 5 */
```

```
x = *stringPointer++; /* example 6 */
```

Example 1 above increments the value of the pointer. The address contained in **stringPointer** changes from 1016 to 1017. Example 2 will use this new address and **x** will contain 69. Example 3 will increment the pointer by 3. **stringPointer** will now be 1020, and example 4 sets **x** to 79 (the contents of location 1020).

The fifth example shows some common pointer manipulations: using the contents-of operator along with pre/post increments and decrements. Here, the operation is a pre-decrement, which causes the value of the character pointer to be **first** reduced by one (to 1019), and **secondly**, the contents of that address are read. The last line shows a post-increment, which causes **first** the current contents of where stringPointer is pointing to to be read, and **secondly**, the address in stringPointer is incremented by 1.

These last two examples show that by doing pre/post increments or decrements, the compiler will choose to manipulate the pointer differently. The pre-decrement moved the pointer backward one byte in memory before accessing the contents of that address. The post-increment accessed the byte first, then incremented the address. This shows that the **order of operations** for pointer variables will follow the same rule that normal variables follow: that pre-increments and pre-decrements will be done first, then the operation (the contents-of operator in this case), and finally post-increments and post-decrements.

**Useful Tip: Pointer Dereferencing**

A good way to think of a dereferenced pointer (or a pointer variable with the asterisk in front of it) is as “go to the location the pointer is pointing to and do something with what is there.”

**Useful Tip: Brackets with dereferenced incremented pointers**

Usually, programmers write the increments and decrements in the following manner:

x = *(stringPointer++);

by putting the increment or decrement inside of brackets when combining them with the contents-of operator. This makes the statement clearer.

## 15.6. Putting It Together

Let's have a look at a C program using pointers:

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

char * capitalizeString (char *string);
char * concatenateString (char *str1, char *str2);

char * capitalizeString (char *string)
{
    while (*string != '\0')
    {
        if ((*string >= 'a') && (*string <= 'z'))
```

```
        {
            /* current character is lower case! */
            *string += ('A' - 'a');
        }/* endif */
        string++;
    } /* end while */
    return string;
}

char * concatenateString (char *str1, char *str2)
{
char *stringStart = NULL;

    stringStart = str1;

    /* get to end of string */
    str1 += strlen (str1);

    /* copy str2 to end of str1 */
    while (*str2 != '\0')
    {
        *str1++ = *str2++;
    }

    /* add new null termination to end of str1 */

    *str1 = '\0';

    return stringStart;
}/* end concatenateString */


int main (void)
{
char s1[80] = "hi!";
char *s2 = NULL;


    s2 = "goodbye!";

    capitalizeString (s1);
    printf ("s1: %s\n", s1);
    concatenateString (s1, s2);

    /* print string using pointer */

    printf ("s1: %s\n", s1);
    return 0;
}/* end main */
```

The first function above, **capitalizeString()** has a single parameter, which is a pointer to a character. Notice how we run through the entire string, character by character. The while loop will loop as long as the contents of the pointer to the string does not equal '\0' - the null termination. As we go, we check to see if the currently pointed to character in the string is a lower case character (between 'a' and 'z' inclusive). If so, we subtract from that character the letter 'a' and add to it 'A', which generates for us an upper case character (remember that subtracting 'a' is subtracting the ASCII equivalent for that character - same goes for adding 'A'). The function is complete as soon as we reach the null termination of the string (ASCII zero), and the current value of the pointer is returned, which is a pointer to the end of the character string.

The while loop will stop as soon as the content of memory is a value of zero, which is the null terminator. Note as well that this is not quite as good a trick as it used to be as modern software must operate in non-ASCII environments (for example Japanese, Chinese, etc.). **Internationalization** of software requires explicit checking for things like terminators, rather than assuming that a terminator is zero. For the purposes of this course, we will assume we are in a North American locale, hence, a termination will be zero, just to keep examples as short and sweet as possible.

The second function is **concatenateString()**, and is used to add one string to the end of the second string. Notice that this function has two character pointers as parameters. The first string pointer will have the second string added to the end of it. We first add to the string pointer the length of the string itself, by calling the standard library function **strlen()**. This gives us a pointer to the end of the string. Next, we simply copy characters one at a time from the second string to the first string. Notice how we increment both pointers as we copy characters. The statement:

```
*str1++ = *str2++;
```

is a fairly familiar one in C whenever data is being copied quickly from one place to another. The C compiler will generate code to extract the current contents of **str2** and place it into memory where **str1** points, then increment both **str1** and **str2**.

This function will place a new null termination into the end of the first character string once the copy is complete. This is needed since we must inform the rest of our software where the new end point of the first string is.

We must always ensure, whenever we manipulate character strings such as the above example, that we always update the null termination accordingly - if we don't we run the chance of having strings 'fly off' into never-never land as functions like **printf()** will never stop until they reach a null terminator character!

## 15.7. Scaling Of Pointer Math

There is one important feature to note about using pointers: depending on the data type the pointer is pointing to, whenever arithmetic operations are performed upon these data types, the arithmetic operation will reflect the size of the data type. For example, when incrementing a **char** pointer, the pointer address is incremented by one to move to the next byte. When decrementing a **short** pointer, the pointer address is decremented by **two**

since an short is two bytes wide. Incrementing a **long** pointer increments the pointer address by 4 since a **long** contains 4 bytes. The same rules apply for **int**, **float** and **double**.

## 15.8. Pointers and Arrays

Pointers and arrays are somewhat interchangeable in terms of how we can access information that can be referenced by them. The notation for arrays (the usual square brackets with an offset inside) is applicable to pointers, in addition to the contents-of operator, because all the array notation does is use the pointer address as the base address of the array. Also, we can use the name of an array as a pointer, because all an array name is is simply a pointer to the first location in the array!

For example, our now familiar character buffer for getting user input is:

```
char buffer[100];
```

We use this array by calling the **gets()** function, as in the following example:

```
printf ("Enter a string of characters ...");
gets (buffer);
```

The **gets()** function is generally prototyped in stdio.h as:

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

Notice that the parameter for **gets()** is a pointer to a character. Why can we pass **buffer** as a parameter to **gets()**? Simple! The array name is the address (a pointer) to the first element in the array. **buffer** is a pointer to the first **char** inside of the 100 **char** array.

The interesting actions we can do with this array stem from the fact that arrays and pointers are somewhat interchangeable. Study the following program:

```
/*
 * PTRDEMO.C
 */

#include <stdio.h>
#include <string.h>

int main (void)
{
char buffer[100] = {0};
char buf2[100] = {0};/* reserve some memory */

    printf ("Enter a string of characters ...");
    gets (buffer);

    printf ("From the 3rd character onwards ...\n");
    printf ("%s\n", buffer + 2);

    printf ("copy what's in buffer to buf2\n");
    strcpy (buf2, buffer);
```

```
    printf ("buf2: %s\n", buf2);

    printf ("copy buffer to buf2 at offset 2\n");
    strcpy (buf2 + 2, buffer);
    printf ("buf2: %s\n", buf2);

    printf ("copy buffer to buf2 at offset 4\n");
    strcpy (&buf2[4], buffer);
    printf ("buf2: %s\n", buf2);
    return 0;
}/* end main */
```

This program uses a standard library function **strcpy()** which is prototyped as follows:

```
char *strcpy (char *s1, char *s2);
```

It will copy whatever is pointed to by **s2** into memory pointed to by **s1**, until a null terminator is located in memory pointed to by **s2**. Only characters will be copied since both parameters are declared as being **char** pointers.

In the **printf()** which says we will display everything from the 3rd character onwards, notice how we specify this fact. Recall that the name of the array is a pointer to the first character. The name plus one then must be the address of the second character, and the name plus two is then the address of the 3rd character. To obtain a pointer to the 3rd character onwards, simply use the address of the array, and add 2 to it! **name + 2** is a pointer to the 3rd character of the array (or the element of the array at offset 2)!

This same idea is used in the next few examples which copy strings of information around using **strcpy()**. The second string copy will copy what's in **buffer** into memory at offset 2 of **buf2**. We generate this pointer location by using the name **buf2** and adding our offset of **2** to it.

But remember that array references can be used as well. **buf2[4]** is an array reference to a character in the array at offset 4. This expression yields the value of a single character at offset 4. To get a pointer to this single value, we could have used **buf2 + 4** as the earlier examples, but we also can use the address-of operator (**&**), and write:

```
&buf2[4]
```

which means "take the address of the character at offset 4 of the array **buf2**". A C programmer has two methods of arriving at the exact same location when working with arrays!

The next example reiterates our earlier discussion about the increment and decrement operators when working with pointers. Improper use of these operators will cause incorrect values to be obtained or stored in memory, leading to corruption or possible lockups of your software.

```
int xx[5] = { 33, 44, -1, 0, -33 };
```

In the above line of C code, we are initializing an array of integers (5 in total) to a certain set of values. The table above shows where in memory this table happens to be stored,

and the values at the expected locations and offsets. Note that this is an integer array, therefore, all addresses are two bytes apart.

```
int *c = NULL;
int y = 0;
int z = 0;
int y1 = 0;
int z1 = 0;

 . . .

c = xx;
y = *++c;
z = *--c;
y1 = *c++;
z1 = *c--;
```

In the above example, we have set the pointer **c** to where **xx** begins in memory. Next, we are using the increment and decrement operators to move the pointer around. The increment and decrement operator affect the pointer in different manners depending on how the operator is placed. (Remember, they will affect **any** variable in these same ways!)

When ++ and -- are in front of a variable, it tells the compiler to increment/decrement the variable **first**, before proceeding to do anything else in the expression. When it comes after, it means increment/decrement the variable **last**, after everything else in the expression is complete.

**y** will be set to the value **44**, because we move **c** up by one integer in memory, then look in memory at that location. **z** will be set to **33** because we move **c** back by one integer before looking in memory.

**y1** will be set to **33** because we look in memory first, then move the pointer ahead by one integer, and **z1** will be set to **44** because we look in memory first, then decrement the pointer.

## *15.9. The scanf() Family*

The way you use the scanf() family of functions is somewhat simple: you supply the function with a formatting string, much like the string given to **printf()**, except that rather than printing values, we are telling the function to extract values of a certain type. We can tell this function to look for integers, characters, strings, longs, doubles; actually, anything that you can print with **printf()**, you can obtain with a **scanf()** family function.

As an example, let's look at the following program: `/* SCANF.C */`

```
#include <stdio.h>

int main (void)
```

```
{
int x = 0;
long a = 0;
long b = 0;
char buffer[100] = {0};


    printf ("enter a string: ");
    scanf ("%s", buffer);

    printf ("\nenter an integer: ");
    scanf (" %d", &x);

    printf ("\nenter 2 long ints: ");
    scanf (" %ld %ld", &a, &b);

    printf ("\nresults: %d %ld %ld %s\n", x, a, b, buffer);
    return 0;
}/* end main */
```

The above program asks us to enter four different pieces of information: a string, an integer, and two long integers. Notice that after the **printf()** calls to print out the prompts to the user, we end up making three calls to **scanf()**, except the formatting string passed as the first parameter is different in each case.

The first **scanf()** is looking for a string, which is signified by the **%s** in the formatting string. **%s** in **scanf()** has the same function as in **printf()**, namely, that a string is about to be manipulated. The function will look for a space-delimited string (i.e. until a space is reached), and place the characters into the array called **buffer**.

The second **scanf()** is supposed to look for an integer. As you can see, the formatting string is **" %d"**, which says to expect an integer. The second parameter is our now familiar **address-of** operator, which will pass the address of where in memory the variable **x** is located. It should be apparent that **scanf()**, when extracting the values asked for, will place them into the appropriate data object, which is supplied by the addresses passed as the second, third, etc parameters. The space before the **%d** is used to tell **scanf()** to skip all **white space** before scanning for a number: this ensures that **scanf()** will ignore spaces or carriage returns already entered from the previous **scanf()**, and get us the value we want.

The final **scanf()** function call is looking for two longs, thus the formatting string is

`" %ld %ld"`

which tells **scanf()** to skip white space, look for a long, skip some more white space and get another long. The two longs will be returned through the addresses of variables **a** and **b** (**&a** and **&b**).

If you're wondering about why you don't pass an address for the string in the first **scanf()**, remember that the name of an array is already a pointer to the memory it occupies! You generate the pointer to the array by simply passing the name of the character array.

The reason we waited until now before using this function is the fact that explaining the necessity of passing addresses of variables to the function **scanf()** would needlessly confuse most neophyte C programmers. Everything you can do with **scanf()** can be done explicitly with **gets()** and the appropriate conversion function (**atoi()**, **atof()**, etc), with much less hassle.

Importantly, **scanf()**, just like **getchar()**, uses an internal keystroke buffer to record keyboard activity. This has implications on how **scanf()** will extract information from this buffer. For instance, if you look for an integer using **%d**, and the user of your software typed in a bunch of alphabetical characters, **scanf()** will fail to extract an integer. Unfortunately, because no processing of the input characters took place due to the data mismatch, the next time you try to extract information, **scanf()** will look at the remaining characters in its keystroke buffer, until it processes the carriage return. This is one of the side effects of the buffering that **scanf()** performs for us.

Unfortunately, that means that scanf() will be rather ill-behaved if you type in anything that is different from what it is expecting. If you have a loop that gets input, it could end up looping forever, getting the same invalid input over and over! This is, obviously, A Bad Thing. That's why it's best not to use scanf(). Instead, you can use a combination of fgets() and sscanf().

## 15.10. Using fgets() and sscanf() for Getting Integer Input

Here's a function that you can use to get integer values from the user in a safer manner. It has an inherent limitation, though, such that it uses a value of -1 to indicate invalid input so you can't tell if the user really entered -1 or if they entered something bad.

```
int getNum(void)
{
char record[121] = {0};
int number = 0;

        /* use fgets() to get a string from the keyboard */
        /* record contains the characters entered on the
         * keyboard */
        /* stdin indicates that we want to get the
         * characters from the keyboard */
        fgets(record, sizeof(record), stdin);

        /* extract the number from the string; put the
         * number just entered into the number variable */
        /* sscanf() returns a number corresponding with the
         * number of items it found in the string; since
         * we're looking for one number, it should
```

```
         * return 1 */
        if( sscanf(record, "%d", &number) != 1 )
        {
               /* if the user did not enter a number
                * recognizable by the system, set number to
                * -1 */
               number = -1;
        }

        /* return the value that we've got */
        return number;
}
```

To avoid the "loop-forever-on-invalid-input" problem inherent to **scanf()**, we can get the entire line from the keyboard all at once and then examine it for valid input.

The act of getting the entire line from the keyboard is done by **fgets()**. You've seen **gets()** so far. **fgets()** is the file version of **gets()**. We can use it in a non-file situation, though, by using a pre-defined variable that represents the keyboard as a file: **stdio**. **fgets()** takes three parameters. The first parameter is the same as you would send to **gets()**: the location of a buffer to put the string in. The second parameter is the reason that we are using **fgets()** instead of **gets()**. It is a number that indicates the upper limit on how many characters (including null-termination) we are willing to get from the keyboard. If we set this second parameter to the size of our buffer, we are guaranteed to never overflow that buffer from this call. Lastly, the third parameter is given by the **stdio** variable that we mentioned before. That tells the function that we want to get the input from the keyboard.

Once we get the string into the buffer, we can scan it using **sscanf()**. **sscanf()** is a variant of **scanf()** that works on strings, not keyboards. That eliminates the problem of invalid input messing you up.

**Course Requirement: scanf()**

Do not use **scanf().** Use **sscanf()** (or one of the conversion functions like **atoi(), atol(),** or **atof()**) instead, in conjunction with **gets()** or **fgets().** Failing to do so will lose you marks!

## 15.11. Common Beginner Errors

- Always make sure that pointer variables have valid values or are **NULL**.
- Be very aware of the difference between, for example, ***myVar**, **myVar**, and **&myVar**.
- When dealing with a string in a loop, be conscious of the need to check for the end of the string (the null-termination).

## 15.12. Summary

This chapter has introduced the concept of the **pointer**, which was defined as a method of accessing individual data objects anywhere in memory through the use of an address. The advantage of such a data type is that we can move around memory very efficiently using simple math on the pointer variable itself. The pointer variables hold an **address**, which points to a particular object in memory. Using the **contents-of** operator, we are able to extract or place information out of or into the memory occupied by the data type the pointer points to.

Pointers can be initialized to point to things like characters strings, since, as we saw with arrays, all a character string is is simply an address which points to the first character of the character array. Pointers can also be generated using the **address-of** operator, which will instruct the C compiler to obtain the address of a data object and place it into the pointer variable. Thus, using pointers, we can instruct a function to be able to pass more than one piece of information back from the function by simply passing the addresses of the data objects to update.

C has a wealth of standard library functions that manipulate strings. In fact, most library functions that require strings as parameters will use pointers internally to get each character one at a time out of the passed string addresses.

Also discussed was the fact that an array and a pointer share similar characteristics. An array name is simply the base address of the array, which is exactly what a pointer is. In this manner, a C programmer can access data within an array using either pointers or array notation.

# Chapter 16. The C Preprocessor

## *16.1. #define*

In this chapter, we attempt to cover a few specific loose ends in our introductory look at the C programming language. The topics covered here do not affect our overall understanding of C, but introduce a few concepts that can make our lives easier when programming in C.

So far, we have used constants by specifying the value of the constant each and every time we used a constant. For instance, the following lines of code illustrate the use of constants as we currently understand them:

```
int a[5] = {0};
int z = 0;
. . .
for (z = 0; z < 5; z++)
{
    a[z] = 0;
}
```

In the above examples, the number 5 is used as a constant in sizing the array and as a loop control condition. If this value of 5 had a special meaning for our software, and later in life, it turned out that each time we had to use the value 5, we should have used the value 6, the programmer will have to re-edit the source code and replace all occurrences of 5 with 6.

This is of course time consuming, tedious and error prone. A better solution is to direct the compiler to use an alphanumeric name to represent a constant, which is accomplished using the **#define** compiler directive.

The **#define** directive allows a programmer to give a name to a constant, and whenever that name is used in the program, a program called the **preprocessor**, which executes prior to the compiler and whose job it is to scan header files and convert all directives into their appropriate values, will replace all occurrences of the name with the constant value it represents. For our above examples using the constant 5, see how a **#define** simplifies coding:

```
#define IMPORTANT_VALUE    5
int a[IMPORTANT_VALUE] = {0};
int z = 0;
. . .
for (z = 0; z < IMPORTANT_VALUE; z++)
{
    a[z] = 0;
}
```

Everywhere we had used the value 5, we now use the name **IMPORTANT_VALUE**, which was declared using the **#define** directive as being equivalent to the constant 5.

In the first statement, the array **a** is being sized to **IMPORTANT_VALUE**, which the preprocessor will replace with the value 5. Using **IMPORTANT_VALUE** does exactly the same thing as using the value 5 directly. Later, to initialize the array to zero, we use a loop control value of **IMPORTANT_VALUE**. The advantage of using the **#define** directive is that if an important constant will change during the course of a program, we can easily change all occurrences by changing the specified value after naming the constant. If we change **IMPORTANT_VALUE** to 10, the array is automatically resized, and the loop will still initialize all values in the array to zero with one small change in our source code.

The general syntax for **#define** is:

```
#define name-of-constant   value
```

An important note: there is **no semicolon** after specifying the value! This is similar to how there is no semicolon after a **#include** line. Also, it is common coding style for professional programmers to try to name constants using totally uppercase letters. In this way, it is easy to spot constants being used in a program.

## *16.2. Macros*

In many software applications, such as spreadsheets and word processing, the software package provides a method of performing commonly repeated keystrokes and mouse-strokes in the form of easy to use **macros**. In C, a macro is a similar concept, and uses the power of the preprocessor software combined with the **#define** directive to allow the compiler to perform some interesting work.

As an example, study the following directive:

```
#define cls()   clrscr()
```

The above directive informs the preprocessor that whenever it sees the text **cls()** in our code, it should replace it with the text **clrscr()**. This is a portable method of defining a clear screen function since not all compilers have a **clrscr()** function. When working with a different compiler, you simply have to change the directive to use the appropriate clear screen function call equivalent (if it exists)!

**Useful Tip: Clearing the screen in Visual C++**

There is no built-in way of clearing the console (or clearing the screen) in Visual C++. An alternative is to use **system("cls");**

Here is another example:

```
#define AVG(x,y)(((x) + (y)) / 2)
```

This is another example of the power of the macro. Here, we've defined a macro called **AVG** which will take two macro parameters, **x** and **y**. The data types for these parameters are irrelevant since all the preprocessor will do is substitute the expression that appears later on the line (a search and replace operation).

To use **AVG**, you may code something similar to:

```
int a = 0;
int b = 0;
int c = 0;
int d = 0;
. . .
a = 5;
b = 9;
c = AVG (a, b);
d = AVG (c, 5 * b);
```

Notice how we simply place values into the parameters of the **AVG** macro, similar to how we pass parameters to a function. The preprocessor will translate the last two statements into:

```
c = (((a) + (b)) / 2);
d = (((c) + (5 * b)) / 2);
```

which is exactly what an average is supposed to do.

The reason programmers use macros sometimes as opposed to function calls is that a function call has associated with it. This overhead is the parameter passing and local variable creation, along with the storage of the return location in memory where the program can continue when the function completes. If a programmer desires to wrap a small computation with a meaningful name such as **AVG**, it may be more beneficial speed-wise to create a macro rather than a function call.

The extra brackets in the **AVG** macro definition ensure that there will be no ambiguity when substitutions are made. This is an essential macro programming concept.

**Useful Tip: Macro brackets**

Always put brackets around the macro "parameters" in the expression.

Commonly found macros include **min()** and **max()**, available in most compiler include files.

## *16.3. Conditional Compilation*

The last topic to cover in this chapter is **conditional compilation**. This is a term given to compiling source code that contains code that should, or should not be compiled depending on the circumstances. For example, during the debugging phase of a project, programmers will place extra code into a program to help debug the danger spots of a program. When the program is deemed to be bug free, normally the debug statements should be removed. If the program is then released, and a new bug is discovered, the programmer may need to add all the debugging statements back into a file. This ends up being a very frustrating process!

The concept of conditional compilation in C is not a new one for C programmers. We have already seen a powerful method of keeping code from being compiled: simply place unwanted code inside of a pair of **comment delimiters**, namely **/*** and ***/**. Any code within the pair of comment delimiters will not be compiled just like any comment text is

not compiled. The C programmer can then simply add comments around debugging statements to keep the compiler from compiling such debugging statements when a program goes to market.

It still becomes a bit of a pain to search through your source code to locate debug statements in order to comment them out. A better solution is to use the conditional compilation directives in C.

To begin a section of code which is to be conditionally compiled, use the following directive:

```
#ifdef CONDITIONAL_FLAG
```

and terminate the section of conditionally compiled code using:

```
#endif
```

The constant **CONDITIONAL_FLAG** can be any name the programmer chooses. If the constant has been defined anywhere prior to the **#ifdef** statement, the compiler **will** compile the conditionally compiled section of code, because the constant has been defined. If not, the compiler will ignore all statements until the **#endif** is located.

The advantage of this is that a programmer can use a specific constant, such as the word **DEBUGGING_IN_PROGRESS** and simply define or remove the definition of this constant whenever switching from a debugging to production mode. The conditionally compiled statements will then either appear or disappear whenever needed, without any effort on the part of the programmer!

Project maintenance software, such as an IDE or MAKE also allow the compiler to be informed that a constant exists even before the file is compiled. At the project maintenance level, flags such as **DEBUGGING_IN_PROGRESS** can be turned on and off whenever a new phase of the project is entered!

As an example, study the following code snippet:

```
#define DEBUGGING_IN_PROGRESS

. . .

#ifdef DEBUGGING_IN_PROGRESS
    printf ("we are debugging\n");
    printf ("variable x: %d\n", x);
#endif
```

In the above snippet, since the constant **DEBUGGING_IN_PROGRESS** has been defined, when the compiler sees the **#ifdef** directive, it knows that the two **printf()** statements between **#ifdef** and **#endif** are to be compiled along with the rest of the software. By removing or commenting out just the line **#define DEBUGGING_IN_PROGRESS**, and recompiling this source code, the two **printf()** statements will not be compiled. This feature should tremendously simplify debugging cycles!

There is a third directive that goes along with **#ifdef** and **#endif**, called **#else**, which allows either one section or another section to be compiled if a conditional compilation flag exists or not. When experimenting with code, a programmer can place new, untested code inside such a conditional compilation statement, and keep the original code in the **#else** portion of the conditional compilation sequence. Whenever the flag is present, the new untested code is compiled, if the flag is not present, the original code will be compiled. This again may help a programmer through debugging sessions.

Related to the #ifdef is a conditional compilation feature called **#if(flag==value)**. The value inside the brackets is a conditional compilation flag, however, the #if will test the flag for a specific value – if the match occurs, then the conditionally flagged code will be compiled, else it will be skipped. This is often how upgrades to software are handled. If you need to recreate an older version of an application, or a lesser featured version, use the #if approach to conditionally compile based on a version number.

**Useful Tip: Fixing compiler errors using #ifdef**

Use #ifdef to help narrow down what is causing compiler errors. Do this by using #ifdef to not compile large blocks of code until the error no longer appears. Then reintroduce small parts of the code block until it reappears again.

## 16.4. Common Beginner Errors

- Don't reflexively put a semicolon at the end of preprocessor directives.

## 16.5. Summary

This chapter has introduced number concepts which enhance our understanding of C. First, we looked at how to reference constants using a name rather than the actual constant value through the compiler directive **#define**. Next, we looked at how the **#define** directive can aid in creating macros - substitutions for simple repetitive code. A macro can aid in making a program more portable between compilers and operating systems. Finally, the topic of conditional compilation was discussed to give the programmer a taste of high powered debugging capabilities in the C language.

# Chapter 17. Structures

## 17.1. What is a Structure?

A structure in C is a data type that the programmer may **create** to help solve a programming problem. This ability makes languages which allow programmer defined data types very popular, since the code you create can be **data driven**, a term given to programs which revolve around the data they will work with, rather than **code driven**, meaning code which revolves around limitations in the language used.

The term structure itself is usually defined in a dictionary as:

**composition, an arrangement of objects, an organization.**

In programming, the idea of a structure relates to the language's ability to be able to organize the primary data types available (**char**, **int**, **long**, etc.) into something which more closely suits the program's needs. For instance, a hypothetical program must keep track of manager names, the number of people they supervise, the amount of their budget and the amount spent by their department. This is an ideal candidate for a structure, as we have four pieces of information that are related to one another.

## 17.2. Structure On Paper

The following is a pictorial representation of the structure discussed above. It is often useful for programmers to organize structures of data in a similar, visual manner before defining a structure in their software.

**Manager Name**

**Number of Subordinates:**

**Budget Allotted:** **$**

**Expenditures:** **$**

If you had 50 of these managers to keep track of, you may have 50 sheets of paper with the above information kept inside of a file folder in your file cabinet.

The above organization is something that can be represented in a C database style program in a number of ways. Currently, we know that if we have to keep track of up to 50 manager names, each no longer than, say, 50 characters, we can create a data object as such:

```
char managerName[50][51];
```

To keep track of the subordinates:

```
unsigned int subordinates[50];
```

and finally, the two financial figures:

```
double budget[50];
double expenditures[50];
```

With the above array declarations, we now have 4 distinctly separate arrays to manage in our C program, even though the 4 arrays are intimately intertwined in their significance.

Even though we have four different arrays to manage, it isn't a totally onerous job to administer such a database. For instance, the following code snippet allows us to enter information into entry 33 of our database:

```
char buffer[100] = {0};
int x = 0;

. . .


x = 33;

. . .

printf ("Enter manager name: ");
gets (managerName[x]);
printf ("Enter number of subordinates: ");
gets (buffer);
subordinates[x] = atoi (buffer);
printf ("Enter budget amount: ");
gets (buffer);
budget[x] = atof (buffer);
printf ("Enter expenditures: ");
gets (buffer);
expenditures[x] = atof (buffer);
```

With just a few lines of C code, we can get information into our database elements.

But what do we do about shuffling information around inside of our database? Currently, in an array implementation as above, we would have to do the following:

```
/* copy entry x into entry y */

strncpy (managerName[y], managerName[x], 50);
subordinates[y] = subordinates[x];
budget[y] = budget[x];
expenditures[y] = expenditures[x];
```

Already you should be able to see that once information is into our array based C program, it begins to become tedious to always be duplicating lines of code to do similar things. It is especially infuriating knowing that the four arrays we are manipulating are actually part of a larger organization, at least on paper. Imagine if we had to keep track of over 100 pieces of information for each database entry!

## 17.3. C Structures

This is where C structures come into play. Thus far we have only been able to create aggregate, or derived data objects based upon the primary data types. With the following new concepts, we will learn how to create a data type which does not **currently** exist in

the C language, and learn how to actually declare derived data objects based upon our own programmer defined data types.

We inform the C compiler we wish to create our own data type through the use of a **structure declaration**. The structure declaration informs C that the declarations between the braces following are **not** variable declarations, but in reality, definitions of the component **fields** that lie within the structure itself.

## 17.4. Structure Notation

The notation for a structure declaration is as follows:

```
struct tag
{
    data-type field1;
    data-type field2;
    . . .
};
```

Once the above declaration is made, a new data type is added to the programmer's repertoire: **struct tag**. Think of the **tag** as the name of the structure. The declaration above does not allocate any memory for the structure; it simply defines a **template** for future variables based upon this template. This template can then be used to declare simple variables, arrays or even pointers.

If we take the example manager database for instance, we see that we are required to define four different fields for our structure. We have a manager name field, maximum 50 characters, plus one for the null terminator. We have a field to keep track of the subordinates of the manager. We have two fields that keep track of the financial performance of a manager. In our earlier approach to this scenario, we simply created four arrays for each of the fields to keep this information for the 50 possible managers.

To demonstrate how to create data types in C, we will define a new data type that can be referenced through the tag **managerStructure,** which will contain the appropriate fields to allow us to manipulate the manager data with ease. We will define this data type using the structure declaration notation described above:

```
struct managerStructure
{
    char name[51];
    unsigned int subordinate;
    double budget, expenditures;
};
```

Notice that we begin with the keyword **struct**, followed by the **tag**, the name the programmer gives the data structure itself. In this case, we have tagged the structure **managerStructure**. This is simply a name to be used in conjunction with the keyword **struct** to allow a C compiler to know which structure you are referring to.

Within the braces of the declaration, you should notice that we have our now four familiar pieces of information for the manager structure. We have declared an array of 51 characters, and given it the name **name**. We then declare an integer with the name

**subordinate**, followed by a declaration of two double precision values, **budget** and **expenditures**.

Although these declarations look remarkably similar to those of a regular variable declaration, they definitely **are not** since they lie between the braces of our structure declaration. Because of the braces, the C compiler can discern between a normal variable declaration and a structure declaration. To complete the structure declaration, we use a closing brace, and a semicolon.

## 17.5. Variable Declarations

So far, the above is only a data type. We are now in the position to actually declare our first variable with the data type we just created. The syntax is as follows:

```
struct managerStructure headManager;
```

The above variable syntax declares a variable called **headManager**, whose data type is **struct managerStructure**. In memory, we have just asked the C compiler to create:

```
[51 bytes for the name][2 bytes for subordinate]
[8 bytes for budget][8 bytes for expenditures]
```

**Useful Tip: Saving Stack Space Again**

Store large local struct variables as static.

## 17.6. Structure Copy

Although it may appear that we could have done the exact same thing by declaring a single array of 50 characters, then a single unsigned integer, and then two double precision variables, we have one very big advantage: you are able to apply the standard assignment operation against structure variables. For instance, we can do the following:

```
struct managerStructure headManager;
struct managerStructure otherManager;

. . .

headManager = otherManager;
```

The last line above will do exactly what is expected: it will update all four fields of the head manager variable with the four fields found inside the other manager variable. It performs an automatic string copy, an integer copy, and two double precision copies **all at once**. This is usually referred to as a **structure copy** or **block copy**.

## 17.7. Field Notation

To actually get information into our structure variable, or to access information already stored in the structure variable, we must have the ability to access the individual fields within the structure. This is accomplished using the structure member notation, which uses the **dot** operator:

```
structure-variable-name.field
```

The structure variable name serves much like the name of an array: it allows the C compiler to generate an address to the start of the structure, and internally in your code, the compiler will generate offsets to each of the fields within the structure in order to access the information stored there. C structures perform all of the addressing work automatically by using the above notation. As an example:

```
printf ("enter manager name: ");
gets (otherManager.name);
headManager.subordinates = 10;
```

The lines above demonstrate effectively the use of the **name.field** approach to accessing the information stored within the structure variable itself.

## 17.8. Aggregate Structures

Returning to our earlier example of a database that must keep track of 50 managers, we again can use our structure approach:

```
struct managerStructure managers[50];

. . .

gets (managers[42].name);
managers[49].subordinates = 10;
managers[3].budget = 100000.00;
managers[0].expenditures = 1234.56;
```

With the lines above, we have created an array of 50 consecutive manager structures in memory, each structure still having the same arrangement of bytes as the individual memory elements. We then set a particular element's fields to the appropriate value.

With this notiation, by using array offset notation, we are referencing a specific manager entry (i.e.: one of the 50 managers in the array), and once we've isolated one particular entry, we can use dot notation to access a field within this one entry. It couldn't be more intuitive than that!

## 17.9. Pointers and Field Notation

We can declare variables such as:

```
struct managerStructure *mgrPtr1 = NULL;
struct managerStructure *mgrPtr2 = NULL;
```

The above declares two pointers to manager structures called **mgrPtr1** and **mgrPtr2**.

Array and pointer manipulations on structures will perform an operation on the entire block of memory occupied by the structure, unless field notation is used.

When a programmer has a pointer to a structure, field notation will use the **arrow** operator:

```
printf (mgrPtr1->name);    /* yes, this will work */
```

Note that an arrow (->) is used instead of a dot. The arrow is simply a minus sign followed by the greater than sign.

As an exercise, prove to yourself that the following two statements are identical:

```
mgrPtr1->budget = 100.22;
(*mgrPtr1).budget = 100.22;
```

Above, the arrow operator references a field within the structure that the pointer variable is addressing. In the second example, we are taking the contents of the memory that the pointer variable is pointing to, which in effect, gives us access to one complete manager structure. Since we have isolated one manager, we can then use the dot operator to access a field within this structure, hence we end up doing the exact same operation as with the arrow operator, albeit in a much more cumbersome manner. Most programmers will agree that arrow operators are much more convenient than contents-of, followed by the dot operator!

## 17.10. Pointer Arithmetic

When moving about in memory using a pointer to a structure, the pointer address math will reflect the size of the structure. To make this point clear, study the following statement:

```
mgrPtr1++;
```

This statement will increment **mgrPtr**, but by how much? If the manager structure takes 69 bytes, then we have just incremented **mgrPtr** by 69! Remember, moving a pointer always manipulates the address stored in the variable **by the size of the data type** the pointer is declared to point to. Since **mgrPtr** is declared to point to an object organized as a **struct managerStructure**, pointer math will manipulate the address stored in this variable by the size of the structure. This is no different than pointers to any of the primary data types. A simple rule of thumb is:

**Pointer manipulations always alter an address by the number of objects, not bytes. The C compiler will determine how many bytes to adjust the address by.**

## 17.11. Putting It Together

Here is a small test program to demonstrate the above concepts:

```
/* STRUCT.C */

#include <stdio.h>
#include <stdlib.h>

struct test
{
    int x;
    char name[20];
    double value;
```

```
};



int main (void)
{
char buffer[100] = {0};
struct test a;

         /* fill a with values */

         printf ("Enter x: ");
         gets (buffer);
         a.x = atoi (buffer);
         printf ("Enter name:");
         gets (a.name);
         printf ("Enter value:");
         gets (buffer);
         a.value = atof (buffer);

         /* output contents of a */

         printf ("x: %d\n", a.x);
         printf ("name: %s\n", a.name);
         printf ("balance: %f\n", a.value);
         return 0;
}    /* end main */
```

## *17.12. Initializing Structures*

Just as we are able to initialize a simple variable, or an array, we can initialize a structure with information at the time we declare the structure. The following example (which initializes the **headManager** variable with the name Bob, 5 subordinates, a budget of $10000 and an expense account of $2332.99) will demonstrate the required syntax:

```
struct managerStructure headManager =
{
    "Bob", 5, 10000.00, 2332.99
};
```

Notice how each of the four pieces of information in the initialization corresponds to the four fields within the **struct managerStructure** organization. When initializing a structure variable, you must match initial data with the correct field, otherwise, the C compiler will abort the compilation with a data type mismatch error.

**Useful Tip: Formatting struct initializations**

It's considered OK to have the entire declaration, including initializers and braces, on the same line if you wish.

## 17.13. Using Structures from the Standard Library

The ANSI C standard library has a number of useful structures already declared for you, and available for use by including the appropriate header files. Here we will study the ANSI C library support for determining the time and date of your computer.

In programming, there is often a need to display the current time that the computer maintains internally. An ANSI standard function called **time()** can be used to ask the computer for the current time, using a special data type called **time_t**. This is a **typedef**ed data type, the significance of which will be discussed in the next chapter. At this moment, it suffices to know that the **time_t** type is identical to an **unsigned long** data type.

The following is the function prototype, declared in **time.h**, for **time()**:

```
time_t time (time_t *t);
```

This declaration shows that **time()** requires a pointer to a **time_t** object, and will return a **time_t** object.

The value returned by **time()** is not of immediate usefulness to a programmer. It is a special, internal, computer representation of the current time. A programmer must take this value, and give it to a number of other functions to extract meaningful information.

The following prototype, declared in **time.h**, is for a function called **localtime()**, which will extract the local time information for a computer. Local time is generally referred to as the time in your time zone. For example, the local time in Toronto will be 3 hours later than the local time in Vancouver at any given moment.

```
struct tm *localtime (time_t *t);
```

**localtime()** will accept a pointer to a **time_t** value, which should be filled in using the **time()** function call discussed above, and return a pointer to a structure called **struct tm**. This structure, declared in **time.h** has the following organization:

```
struct tm
{
    int   tm_sec;    /* seconds, 0 - 59 */
    int   tm_min;    /* minutes, 0 - 59 */
    int   tm_hour;   /* hour, 0 - 23 */
    int   tm_mday;   /* day of the month, 1 - 31 */
    int   tm_mon;    /* month, 0 - 11 */
    int   tm_year;   /* years from 1900 */
    int   tm_wday;   /* days since Sunday, 0 - 6 */
    int   tm_yday;   /* days since Jan 1, 0 - 365 */
    int   tm_isdst;  /* daylight savings if non-zero */
};
```

This is an elaborate structure, which gives you everything you need to know about the computer's view of the time and date.

Another useful ANSI C function is **asctime()**, which will generate a text version of the time, based upon information inside of a **struct tm** variable. The prototype is:

```
char *asctime (struct tm *t);
```

This function will also require the **time.h** header file. The return value is a character pointer, which will point to a buffer of text, organized as following example output of **asctime()**:

*"Wed Jun 15 12:37:22 1994\n"*

This function gives your software a simple method of obtaining a printable representation of the time your computer has. The following program will tie these functions together to print the time and date your computer currently is maintaining:

```
/* ansitime.c */

#include <stdio.h>
#include <time.h>

int main (void)
{
time_t theTime;
struct tm *t;
char *ptr = NULL;


         time (&theTime);
         t = localtime (&theTime);
         ptr = asctime (t);

         printf ("The time and date is: %s\n", ptr);
         printf ("The day of the year is: %d\n",
                              t->tm_yday);
         return 0;
}     /* end main */
```

The program will first obtain the system time, using a call to **time()**. The parameter is the address of the variable **theTime**, which is of type **time_t**. **time()** will fill in the current time, allowing us to extract the local time information, by calling **localtime()**. The returned pointer, of type **struct tm** is then given to **asctime()**, which will convert the structure into a human readable string. We print this string of characters in the first **printf()**, and the second will output which day of the year it is.

## 17.14. sizeof

An interesting and useful feature of the C language is the **operator** called **sizeof**. It is called a operator because, whenever it is used, the C compiler will not generate code to call a function, rather, it fills in the actual size of the data object following it instead.

**sizeof** can be thought of as a constant which happens to equal the number of bytes of the data object following it. For example:

```
double things[10] = {0.0};

. . .

printf ("size of array: %d\n", sizeof (things));
printf ("size of manager structure: %d\n", sizeof (struct
managerStructure));
```

The first line will display the number of bytes inside of the array called **things**. Since **things** consists of 10 consecutive **double** data objects, this line will display a result of 80 (10 objects, each object 8 bytes long). The second will display the number of bytes in our manager structure, which will cause 69 to be printed. **sizeof** will become very useful when we discuss dynamic memory, and file handling.

**Useful Tip: sizeof**

You can use sizeof with both variable names and data types.

If you are doing **sizeof** in conjunction with a data type that is more than one word long (e.g. a struct), put the data type in brackets.

## 17.15. Bitfield Structures

Another variation on structures involves using just parts of bytes store information more compactly. A bitfield structure is a special type of structure which allows a programmer to define and give field names to bits within an integer. For example, a bitfield structure for date and time bitfields include:

```
typedef struct time_bf
{
    unsigned int sec: 5;
    unsigned int min: 6;
    unsigned int hour: 5;
} TIME;


typedef struct date_bf
{
    unsigned int day: 5;
    unsigned int month: 4;
    unsigned int year: 7;
} DATE;
```

Notice that in a bitfield structure, you can specify the size of a portion of a bitfield structure, as well as the data type. In the example above, we declare that the day field will take up 5 bits, the month field will take up 4 bits, and the year field will take up 7 bits.

Using the bitfield fields is very easy. Simply treat them as normal structure fields.

## 17.16. Summary

In this chapter, we introduced the ability in C to create programmer defined data types. This ability allows a C programmer to create data objects that will enable a simple solution to a programming problem, rather than trying to code around the limitations of a language.

There are two parts to declaring a variable to be of a programmer-defined type. The programmer must first define to the C compiler the fact that a data structure is to be used in a program. This is done through the use of the **structure declaration**. The structure declaration is nothing more than a declaration of the template, or organization of the data structure. After declaring the structure, the programmer has the freedom to create variables of that newly defined data type, as if the data type had been in existence forever. Simple variables, pointers and arrays can be created based upon programmer-defined structures.

Structures simplify work for a C programmer because they declare variables that contain the appropriate fields of information as a single unit. A simple assignment statement using structures has the ability to copy many bytes of data from one location to another location in memory with almost no effort on the part of a programmer.

The pieces of data that are stored in a C structure are referred to as **fields**, and using field notation, a programmer can obtain access to the individual pieces of information a structure variable contains. Thus, a structure variable can be manipulated as a single unit, or can be accessed right to the level of the constituent fields, offering tremendous flexibility and potential to a C programmer.

We looked at the operator **sizeof**, which can be used to compute the size of any data object, be it a variable or data type. We also looked at the use of bitfields.

# Chapter 18. Unions, Enumerations and Type Definitions

## 18.1. Unions

We've learned how to tell the C compiler that we wish to write code to manipulate data types that are not normally present in the language itself. This is done using the structure declaration syntax, and it allows the C programmer to be able to define a data structure to suit the needs of the data they are manipulating, rather than rework the data to suit the programming language.

Another approach in C that allows a programmer to define new data types is the **union**. The union is similar to a structure in terms of declaration syntax, but they way C generates code to access the members of a union is vastly different.

A union is defined as a block of memory that has been designated to be able to hold a number of different data types **all** at the same time. Technically, a block of memory cannot hold, say, both a **float** and a **char** at the exact same instant, but there are times that a programmer needs the flexibility to reassign storage within a block of memory to a different data type to suit the immediate needs of a program. This may sound like a very far-fetched idea, but with a couple of examples, the confusion should diminish.

## 18.2. Union Notation

The declaration syntax, as stated earlier, is almost identical to that of a structure declaration:

```
union union-name
{
    data-type field-name1;
    data-type field-name2;
    . . .
};
```

The only difference is that we use the **union** keyword rather than the **struct** keyword. Let's see an actual real-life example of a union:

```
union stats
{
    int rbi;
    double era;
};

struct playerInfo
{
    char playerName[51];
    int position;
    union stats vitalStatistics;
};

struct playerInfo player1;
```

Notice that, in the above examples, we have declared a union **union stats** and a structure **struct playerInfo**. The union we have declared is meant to hold, in the same memory block, either a runs batted in (RBI) total (useful for all players except the pitcher), or the earned run average (ERA) figure (useful only for a pitcher). If you are familiar with baseball, you will know that, in the American League, everyone but the pitcher will have an RBI total, and only the pitcher will have an ERA value. This is an example of two **mutually exclusive** pieces of information: never shall the two pieces of information interfere with one another, resulting in an "either one or the other" type of situation.

The question may arise at this time: "why would you want to store either one type of information or the other?" Simple! Rather than declaring one structure to hold information for a pitcher, and separate structure to hold the information for a batter, take advantage of the fact that neither the twain shall meet, and have the C compiler generate code to handle the mutually exclusive information.

The structure declared above is fairly simple. An array of 51 characters will be allocated, followed by an integer that will hold the position number for a particular player, followed by the union. An example of a position number scheme may be:

```
#define CATCHER 1
#define FIRST_BASE 2

. . .

#define PITCHER 9
```

Within our structure, we store the information needed to differentiate amongst the different positions a player may play on a team, which is the purpose of these definitions.

We could have designed our structure to have a double precision floating point value as well as an integer to record the RBI or ERA totals. But is this approach feasible when you have to record all the statistics for all the players in the major leagues? You will be wasting a **double** for each non-pitcher, and wasting an **int** for every pitcher in the database: memory that will quickly add up with large databases.

The union, therefore, gives us the ability to access a block of memory in various fashions. With our union **union stats**, we see that in the same memory, we can hold either the **double** or the **int**. In this manner, we know that if our variable **player1** has the **position** field set to 9 (pitcher), we can write our software to extract the double precision floating point value for the ERA. If the position is anything else except pitcher (9), we can write our software to instead extract the RBI total out of that same memory block - hence, the same memory block does double duty here! In our statistics database example, this has the ability to save memory since we no longer require to store two separate pieces of information: only save the one that makes sense.

The memory usage for the union declared above is:

Table 18-1: Union Offsets

| | |
|---|---|
| offset 0 | byte 0 of **int *OR*** byte 0 of **double** |
| offset 1 | byte 1 of **int *OR*** byte 1 of **double** |
| offset 2 | byte 2 of **double** |
| offset 3 | byte 3 of **double** |
| offset 4 | byte 4 of **double** |
| offset 5 | byte 5 of **double** |
| offset 6 | byte 6 of **double** |
| offset 7 | byte 7 of **double** |

The union places the 2 bytes for the **int** over top of the first two bytes of the **double**, using the space effectively since we don't need both pieces of information at the exact same time in our statistics example.

As should be plain from the above example, unions can be used to save memory when working with multiple structures that have many common fields. In situations where only one or the other field is needed, such as our statistics example above, a union is ideal. By overlaying these fields one on top of another in a union, we have the ability to dynamically change the type of a portion of memory at will, without resorting to odd casting syntax or performing unorthodox operations with pointers.

## *18.3. Union Field Access*

The syntax for accessing fields in union is identical to how you access fields in a structure:

```
if (player1.position == 9)
    player1.vitalStatistics.era = 0.325;
else
    player1.vitalStatistics.rbi = 121;
```

In the above two statements, notice that to access either of the two fields within our union, we have used the dot operator. The first dot, after the variable name, is used to access the field within the structure (this field is called **vitalStatistics**, which is defined as a **union** in this case). The second dot is used to access the field within the union itself. This is a good example of a nested structure/union, as one structure/union is embedded within another.

Pointer notation is exactly the same for unions as is for structures. A pointer to a union is simply a pointer to a memory block that can contain more than one representation of data at one time.

**Useful Tip: Union Portability**

Please note that unions are often inherently non-portable between different brands of microprocessors. For instance, a union between an array of 2 characters and a short integer will compile cleanly on both a Macintosh and a PC compatible computer. On the PC, when storing a short integer into the union, the low byte is stored first, then the high byte. On the Macintosh, the high byte is stored first, followed by the low byte.

Code that is written to extract the high and low bytes of an integer **will not** access the same portions of the integer on the Macintosh as it did on the PC. This is not a fault of C, rather, it is a problem that comes up in the internal hardware of different microprocessors on the market. The idea of the union is fully valid, just its portability is questionable.

In case you are wondering, this issue is known as the “big endian” versus “little endian” problem in memory organizations. There is no right or wrong way, just different ways!

## *18.4. Enumerations*

Another programmer defined data type that is available in the C programming language is the **enumeration**. This is a interesting data type, as its basic purpose is to create automatically numbered constants for the programmer.

Often a programmer must work with a set of numbers which are to represent different objects. For instance, if a programmer was to work with the days of the week, a programmer may define the days of the week as such:

```
#define MONDAY      1
#define TUESDAY     2
#define WEDNESDAY   3
#define THURSDAY    4
#define FRIDAY      5
#define SATURDAY    6
#define SUNDAY      7
```

The programmer created the above list of names, and it was the responsibility of the programmer to define values to accompany the names above. There are many such examples of lists that need a numeric constant in order to represent them, such as colours, names of months, and so on.

Sometimes a programmer is faced with the prospect of adding in a new item to such as list. For instance, what if a new day (called for the lack of a better name, NEWDAY) was added in between MONDAY and TUESDAY? C code must be altered to reflect that MONDAY is earlier than NEWDAY, and that TUESDAY is after NEWDAY. The new list would become:

```
#define MONDAY       1
```

```
#define NEWDAY        2
#define TUESDAY       3
#define WEDNESDAY     4
#define THURSDAY      5
#define FRIDAY        6
#define SATURDAY      7
#define SUNDAY        8
```

Notice that we had to change the values assigned to TUESDAY through SUNDAY all because of the addition of NEWDAY. Although editors found on most C compilers these days would make the corrections reasonably simple, renumbering a list of 1000 different names would become **very** tedious.

This is where the enumeration comes into play. An enumeration instructs the C compiler to automatically generate a list of numbers for a set of data objects the programmer will create. This numbering action is done at compilation time, and the programmer need not worry about what numbers are assigned: this is the C compiler's job. The compiler will take the objects within an enumeration and increment the value given an object by one as it compiles each object in the list. The first object is given a value of 0, the second has a value of 1, and so on, unless the programmer overrides this feature. Hence, the value of the lastest entry in an enumeration listing is the previous value, plus 1.

## 18.5. Enumeration Notation

The syntax of an enumeration is quite simple:

```
enum enumeration-name
{
     enumerated-object,
     enumerated-object = override-value,
     enumerated-object,
     . . .
};
```

The second object in the list above is set equal to a special value. This is the **override** feature of enumerations: you can override, or specify values for certain objects. For the next object, the compiler will simply increment the previously specified value, which in this case was the override value, to generate the next value.

An example of a valid enumeration based upon our example above:

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

With the above enumeration declaration, the C compiler would automatically assign the numbers 0 through 6 to MONDAY through SUNDAY. If we now had to add NEWDAY in between MONDAY and TUESDAY, we simply add it to our enumerated list, and voila, the C compiler would renumber the list for us with no effort on the C programmers part whatsoever! The resulting enumeration would be:

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

An added advantage of the enumeration is the ability to declare variables of type **enum name-of-enum**. This makes it plain to the reader of a C program that a variable is only intended to hold a certain subset of values. We can declare the following:

```
enum daysOfTheWeek weekday;
```

and perform the following code:

```
weekday = WEDNESDAY;
if (weekday > TUESDAY)
     printf ("past the halfway mark of the week\n");
```

Hopefully the above examples show the power of the C compiler in doing much of the work for the programmer as possible. Small features such as this make C the important and exciting language that it is.

**Useful Tip: enum values**

You can specify the actual value for individual elements of an enum if you want.

## 18.6. typedef

When naming structures, unions and enumerations, a shortcut a C programmer may take is to use the **typedef** keyword. The **typedef** is an instruction to the compiler that whenever a **typedef**ed name is seen, substitute the (normally large and hard to type!) proper definition in its place. It is similar to a **#define** statement, but has much more powerful substitution capabilities and allows proper type checking to be done by the compiler when declaring variables, parameters, etc.

The generic syntax for a type definition is as follows:

```
typedef existing-data-type new-type-name;
```

To explain this concept, it is best to just use an example:

```
typedef struct testStructure
```

```
{
    int a;
    long b;
    double c;
} TEST_STRUCT;

TEST_STRUCT test_variable;
```

What we've done above is to declare a structure called **struct test_structure**, but because we have placed the **typedef** keyword in front of it, we are actually telling the C compiler that we are about to give **struct testStructure** a much simpler name. The simple name we have given this structure is **TEST_STRUCT**, and we can now use **TEST_STRUCT** as a type declarator just like **int** or **float**, etc.

Programmers often create typedef'ed names in upper case, to help visually distinguish them from other data types in your application - a visual reminder that you've made a redefinition. You will see other such examples as you proceed through the remainder of this course!

The **typedef** can be used to redefine a type name for **unions**, **enums**, or just about anything in the language. In fact, **typedef** is one method that is used to make software portable across different computing platforms by taking the machine specific type declarations (such as **int**) and giving them new names. A **typedef** can also make experimentation with data types simple because the defined data type can be changed once inside the **typedef** rather than throughout all occurrences of the original data type. In either case, a simple change to the **typedef** name will allow variables declared through the **typedef** to assume a different data type. For example, the following are examples of **typedef**s commonly found in C code to allow simple software maintenance across different target machines:

```
typedef int WORD;
typedef unsigned int UWORD;
```

A programmer may declare a number of variables to be of type **WORD** or **UWORD**, and the programmer now has the ability to easily change these variables to another data type without resorting to searching and replacing all occurrences of **int** to another data type. Again, let the compiler do the work for you!

As another example, we saw in the previous chapter, the ANSI C library defines a data type called **time_t**. This is a simple **typedef** for a long integer:

```
typedef unsigned long time_t;
```

Writing software for operating systems such as Microsoft Windows will uncover the use of hundreds of different **typedef**ed data types, all designed to easily port current Windows application code to more advanced versions of this operating system.

## 18.7. Common Beginner Errors

- Make sure that your use of the union actually makes sense. There might be a better way to do it (especially if you're using C++).

## 18.8. Summary

In this chapter we explored two other programmer defined data types: the **union** and the **enumeration**. The union looks much like a structure in terms of its declaration, but the code produced by a compiler to work with unions is vastly different. The union is a method of being able to extract more than one type of data object from a common memory block. The union is most often used to make a flexible data structure since a field can be of one type for one application, and another type for a different application.

The enumeration is a declaration in C that instructs the compiler to enumerate, or number, a list of constant declarations. The compiler is responsible to generate unique values for a list of objects, saving much time over the traditional method of using the **#define** constant declarations.

Finally, we had a look at the **typedef**, which is a directive to replace one declaration with another declaration. It is useful in shortening names of structures and unions, as well as allowing portability and experimentation in software.

# Chapter 19. Multi-file Projects

## 19.1. Introductory Notes

In this chapter, we will look at a few items that may make C programming simpler, especially when it concerns a large programming project.

Currently, we have been writing software whose source code is contained completely inside of a single source file, except for a number of standard header files which we include. There are times that a programmer may want to split up a source file into a number of smaller, more manageable pieces. This chapter will explore the concepts behind multi-file compilation, which most C compilers refer to as **projects** or **workspaces**.

As well, we will look into creating your own header files. Finally, we will look again at a topic which has been bantered about: conditional compilation.

## 19.2. Projects and Workspaces

The term 'project' or ‘workspace’ is given data which is composed of a number of related C source code files. The IDE will take the project and create an executable file after linking the object files from these multiple source files together. The compiler must be aware of the files in a project, and must be able to create object files for each of these files before the linker even has a chance of attempting to link together an executable file.

The main reason a programmer will choose to make a project, as opposed to a single C file program, is that as C programs grow in size, it takes longer and longer to compile the long program. Imagine a program that is 100000 lines long, taking 10 minutes to compile. Imagine then that a programmer inadvertently has a syntax error on the 99999th line of this file. To fix this, the entire file must be loaded into a text editor, and then the syntax error must be fixed, and then the entire file must be recompiled. The main drawback behind this is that 99.9% of this file **did not** need recompilation, only that last bit of code, hence, an enormous amount of time was wasted in this exercise.

To speed up production of executable code, professional programmers will split large C files into smaller, more manageable pieces, and use a project management system to get the compiler to recompile **only** those C files that have changed since the last **build** of the executable. By doing so, the amount of time needed to recompile the executable drops considerably, as only the changed C files are actually processed by the compiler - the untouched C files already have valid object modules on disk for the linker to use!

## 19.3. Variable Declarations

There are a number of issues a C programmer must be aware of before attempting to work with multiple files. Firstly, when using global variables, it is highly advisable to create a separate file to contain all globals for a multi-file project. This separate file will then be compiled by the compiler in order to create the static lifetime storage the project executable requires. The other files that may reference these globals, however, must

know about the different globals that have been declared. There is a special keyword in C which allows a C compiler to treat a variable declaration as an indication that the variable has been properly defined in another file. The keyword is **extern**, and it is prepended in front of the type declaration for a variable to inform the C compiler that the global being declared has been formally defined in another file.

For instance, let us assume the file FOO.C has the following globals defined:

```
unsigned int variable1;
long anArray[10];
double balance;
```

If a second file in the project, called BAR.C, is required to access the global variable **balance**, BAR.C would make a declaration such as the following:

```
extern double balance;
```

The declaration above tells the C compiler that the variable **balance** is a double precision floating point value, but the storage for this variable has been declared in another file. Thus, no storage is allocated when using the **extern** keyword, as its only purpose is to allow the compiler to know the data type of an object that already exists in another component of the project.

Again, this is an extremely important concept to grasp. The C compiler will gleefully reserve global variable space for your program every time it comes across a global variable declaration. However, if two or more global variable declarations with the same name exist when the linker links your object modules together, the linker will exit with an error, informing you of duplicate global variable declarations. The **extern** keyword rids this error from your project!

To continue with this example, if a third file in the project, TEST.C, requires the variable **variable1** as well as the array **anArray**, TEST.C would make these declarations:

```
extern unsigned int variable1;
extern long anArray[10];
```

Notice that the **extern** declarations are virtually identical to the standard declarations, so with a good editor, simply copy the globals needed from one file to another, and insert the keyword **extern** at the start of each declaration.

Note as well that if each file in a five file project requires a global variable, then each file in the project **must** include an external declaration as we've seen above. The more files in a project, the more tedious this process becomes, as you should hopefully be noticing. A remedy to this situation will be described shortly.

## *19.4. Prototyping*

In the case of a multi-file project, there are circumstances which require a programmer to prototype a function which exists in one file and is to be used in another. The use of the prototype allows the C compiler to know the parameter types, as well as the return type of the function, enabling the compiler to generate the correct code to call the function.

The format for a function prototype is almost identical for a function declaration. Recall, for example, this function from earlier:

```
char *
concatenateString (char *first, char *second)
{
    /* code for this function */

    . . .

    return first;
}    /* end concatenateString */
```

This function will have a function prototype which looks like this:

```
char *concatenateString (char *, char *);
```

If the above function is required in all five files of a project, the prototype **must** be repeated in each of the five files, much like how global variables must be **extern**ally defined in each file that requires use of the global.

## 19.5. Common Constants

Again, like the situations regarding common functions and common global variables, constants which are declared in one file and used in many other files of a project must be redeclared for each and every file that uses it. If FOO.C uses:

```
#define TEST_VALUE          100
```

and so does BAR.C, TEST.C and FILE.C, then each of the last three files above must have the same declaration so that the compiler will understand what to do with the name **TEST_VALUE**. This becomes a tedious and error prone situation to repeat constant declarations over and over again for a multiple file project.

## 19.6. Programmer Defined Header Files

With the above problems described, the question that is on all programmers minds at this moment is "why have a multiple line project at all"? The answer is: **program maintainability**. If everything is squashed into a huge, 100000 line file, a programmer who must maintain that software will cringe each and every time a change must be made, since it will take hours just to locate the appropriate lines that the change must be made on, not to mention the fact that it takes a fair bit of time to compile a monster source code file such as the one described. But again, if we have so many problems with multi-file projects, isn't one evil better than the other?

There is a solution to the above problem which makes the maintenance of multiple file projects a breeze. Programmers have the ability to create their own **header** files, in the same spirit as headers like **stdio.h** and **string.h**.

A header file is simply a chunk of C code which the compiler will compile when the **#include** directive is located by the compiler's preprocessor. A programmer needs to add common definitions such as external globals, prototypes, structure declarations and

constants to a header file, and include that header into each file of a multi-file project, and the problems described above are solved forever!

For example, study the following listing:

The file **TEST.H** is a programmer defined header file since it does not exist as part of the standard collection of header files for C compilers. By including **TEST.H** with each file in a multi-file project, each source code file will know that three global variables exist, that a constant called **TEST_VALUE** has been defined, as well as knowing the exact parameters and return value for a function called **concatenateString**. Note that we may even include headers from within a header, which leads to the common practice of placing the **#include** for common header files inside of a programmer defined header file.

You may be wondering what the strange statement using the name __TEST_H__ is all about. This is a professional programming feature, which prevents the accidental recompilation of this header file more than once. Compiling a header file multiple times in the same compile sometimes causes problems. By checking to see if this flag is defined, the compiler can ensure that the file is compiled the first time it is included, and after that point, due to the empty definition of __TEST_H__ at the end of this file, the file will not be compiled again. Again, this is not a crucial feature, but a welcome one in large, hard to manage projects. See below for more details on conditional compilation.

```
/* TEST.H */

#ifdef __TEST_H__
#define __TEST_H__

/* common standard headers */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* globals */
extern unsigned int variable1;
extern long anArray[10];
extern double balance;

/* prototypes */
char *concatenateString (char *, char *);

/* constants */
#define TEST_VALUE         100

#endif
```

For your compiler to access your header file, a slightly modified version of **#include** is used:

```
#include "your_header.h"
```

Notice the use of double quotes. The quotes tell the compiler to look in the current working folder for the header file, and if not located there, it should look in the normal location for system header files. The angle brackets that are normally used inform the compiler to *only* look at the system header file directory.

Large programming projects use this feature of the C language religiously. There is almost no reason to redefine large numbers of previously defined values, variables and prototypes since a programmer can do it once in header. The header files a programmer creates can be thought of as a great labour saving feature of C!

## 19.7. When Will A Compiler Compile A File?

A compiler will always try to keep the executable file as **fresh** as possible. This means that when doing a compilation, the compiler will attempt to check and make sure all files that are in need of compilation are compiled prior to linking object files together.

To be more exact, the control program for the compiler is responsible for doing the checking. IDEs will always check to see what files need compilation before attempting any link operation, unless the programmer overrides this checking feature. In most other compiler packages, especially UNIX based compilers, a special program called **MAKE** is invoked whenever a project requires compilation. The MAKE program will check, using a sequence of rules, as to which files need compilation before proceeding to the link phase.

Both the IDE and MAKE can be instructed to compile a file if a header file for that file is changed, since making changes to constants or variable declarations within header files is a more than enough reason to recompile a file that depends on that header. Header files are normally called **dependencies** for a source code file.

```
# makefile example

foo.exe: foo.obj bar.obj
    cc foo.obj bar.obj -o foo.exe
foo.obj: foo.c test.h
    cc -c foo.c
bar.obj: bar.c test.h
    cc -c bar.c
```

Consult the help pages for your compiler to discover how to make your compilation package understand you are creating a multi-file project. Again, each different compiler manufacturer treats this topic in a slightly different, and slightly incompatible manner!

A typical MAKE instruction file (called, naturally, a **makefile**) appears as follows:

To decipher this file, note that we have an executable file identified on the left side of a colon. This is the **target** of the makefile. Next, we specify after the colon the files that foo.exe depends upon. These are called **dependencies**. The next line informs the MAKE program what command will create foo.exe when the dependencies show that a rebuild is necessary. Here we are invoking a UNIX oriented **cc** C compiler.

Above, bar.obj is dependant on bar.c and test.h. If either of those files are newer than the currently existing bar.obj file, then bar.obj will be recompiled using the command line shown.

foo.exe requires that foo.obj and bar.obj be up to date before it will be linked. By properly specifying dependencies, if bar.obj has just been created, then foo.exe will need to be relinked to update it to the latest version.

Think of a **makefile** being nothing more than a recipe for a compiler and linker to keep an executable up to date. With large, multi-project files, nothing can beat a **makefile** for ensuring that a project is delivered in a repeatable, consistent manner.

## 19.8. Guidelines for Splitting Files

Here are some general guidelines for determining how to split files in large projects:

- split .c files across general topic areas
- put all prototypes in .h files
    - .h files including prototypes will most likely be included in all .c files
- put all global variables in a separate .c file
    - put extern references to those global variables in a separate .h file
- put data types and constants that are used in multiple source files into separate .h files

## 19.9. Example of Splitting Files

Let's say that you had a program that read in formatted financial data from a file, filtered and summarized it, and created a summary output file. Here's one way (of many!) that you could split the files:

- input_format.h: contains the struct definition and constants dealing with the formatting of the input file
- protos.h: contains prototypes
    - this file will most likely have a line including input_format.h
- main.c: containing main() and not much else
    - this file will have lines including protos.h and (most likely) input_format.h
- get_input.c: containing all functions concerned with getting the data in from the file
    - this file will have lines including protos.h and input_format.h
- process_data.c: containing all functions concerned with processing the data once in
    - this file will have a line including protos.h
- create_output.c: containing all functions concerned with filtering and outputting the data
    - this file will have a line including protos.h

## 19.10. Common Beginner Errors

- Avoid reusing already existing projects. You might end up with two versions of main() (one from your old source file and one from your new one).
- Make sure that you set up a Win32 console application.
- Separate include and source files appropriately.

## 19.11. Summary

In this chapter we explored a number of concepts which allow some sophisticated programming to be accomplished. Firstly, the topic of multiple file projects was discussed. Basically, a multiple file project is no different from a single source file C program except that some precautions must be taken. Any globals must be redeclared as **extern**al to a file, since the compiler will create errors if more than one global by the same name exists, even in a multi-file project.

As well, function prototypes, structures, unions and constants must be redeclared for each file that requires them. A simpler approach to this problem is the use of a programmer defined header file, which contains important information for each file in a project.

A brief discussion was held on the concepts behind how the control software for a compiler determines how a project will be compiled. This included development environments and the MAKE program.

# Chapter 20. Dynamic Memory

## 20.1. Static Memory Allocation

In our programming to this point, we have worked with two types of memory allocation for our variables: variables which are allocated when our software begins execution and variables created when we need them and disappear when we no longer need them.

The C compiler, when generating the code for our program, will also generate the code to create and destroy local variables as we enter and exit functions, as well as creating and destroying global variables when our software begins and finishes. Because it is the C compiler doing this work, many programmers do not realize that memory is being **allocated** for the purposes of our software. Recall that local variables are allocated their space once a function begins. Once the function is complete, the variables are no longer needed, and the memory they occupied is released back to the operating system that executes our software.

Static memory allocation is the term given to memory that is allocated for static (automatically created) data objects. A global variable, local variable, or a table of values, are good examples of statically allocated memory. The following shows a statically allocated block of memory:

```
struct managerStructure managers[10];
```

If the above is a global or static variable, the memory, which amounts to **sizeof (struct managerStructure) * 10** bytes, is obtained when the software begins, and will disappear when the software has completed execution. Again, it is the responsibility of the C compiler to generate the code to create these variables, as well as cleaning up memory to remove their presence. If the above variable was a local variable, the stack would be accessed to allocate the memory upon function call, and clean up the memory upon function exit - again, the work is done for you by the C compiler.

It is interesting to note that a programmer will always know how much memory has been statically allocated by adding up the number of bytes for all global and static variables, arrays, pointers, and so on.

## 20.2. Dynamic Memory Allocation

The term dynamic memory allocation is used by programmers to denote the action of obtaining memory for data storage **only when required**, or on a **dynamic** basis. This is obviously different from static memory allocation, where we know ahead of time how much memory is required. In dynamic memory schemes, the programmer may not know ahead of time how much memory an application will require, hence, a new scheme must be found to allow memory requirements to grow as time passes.

We have been exposed to a very simple and limited variation of dynamic memory allocation, in the form of local variables. Although these variables are considered statically allocated variables, they nonetheless exhibit some of the basic properties of dynamic memory: these variables do not occupy memory until they are needed (until a

function is called). These variables will also remove themselves from memory when they are no longer needed (after a function is complete).

Windows and UNIX both manage memory effectively, and encourage programmers to write software so that memory can be shared properly between applications. Could you imagine the problems in using Windows if a word processor ate up all memory, and thus the operating system would disallow running your favourite spreadsheet at the same time? Obviously, word processors, spreadsheets, etc., must be doing something to ensure memory is always available for other programs.

But how do we judge how much memory one needs? This is where some data analysis comes in handy. If you are writing a database that must keep track of, say, 10 objects, it will be much simpler to allocate static storage for these 10 objects since the amount of memory required to store only 10 objects is rather small compared to the amount of free memory in your system.

But what happens if you have to store up to 1000? By statically allocating an array of 1000 objects, you will begin to run out of memory, or at least reduce the amount of free memory to a minimum, leaving too little for other operations. This is where dynamic memory allocation comes in handy, and where pointers to structures play an important role.

Let us use the **struct managerStructure** example again. If you didn't want to allocate storage for an array of 10 entries of this data type until you really needed it, you must at least declare a variable that will point to a future piece of memory for this structure.

Hence, a dynamic memory scheme will always use pointers in some fashion - an array of pointers to each data object allocated, or perhaps a single pointer to a dynamically resizable block of memory! In this example, you may create an array something to the effect of:

```
struct managerStructure *managers[10] = {NULL};
```

The line above will declare an array of 10 *pointers* to manager structures. We have not yet obtained space for the 10 individual structures, therefore the amount of memory used so far is only **10 * sizeof (struct managerStructure *)** bytes (10 * 4 or 40 bytes). This is considerably less than allocating 10 complete manager structures, which would use up over 600 bytes of memory.

Of course, now the question begs ... "how do we allocate memory when we need it?" ... read on ...

## 20.3. malloc(), calloc() and realloc()

When we come to a point where we actually need to store information for a manager, we need to **dynamically allocate**, or tell the operating system that we now require memory. This is done using the following function calls, prototyped in **stdlib.h**:

```
void *malloc (unsigned int size);
void *calloc (unsigned int numElements, unsigned int
sizeOfElement);
```

```
void *realloc (void *old, unsigned int newSize);
Now we begin to get into some tricky stuff!
```

The notation **void *** simply denotes that these functions return a **generic pointer**, (a pointer to **something**), and as a programmer, you are required to **cast** the return value into a pointer of the appropriate type, otherwise endure warnings about data type mismatches from the C compiler.

**malloc()** is the ANSI C standard memory allocation function, which will obtain from the operating system a **block** of memory of **size** bytes, and return a **generic pointer** (address) to that block of memory. **calloc()** is used to obtain more than one consecutive block of memory of a specific size. You could, theoretically, dynamically allocate all 10 manager structures using a single call to **calloc()**. **realloc()** is used to alter the size of an **already** allocated block of memory. This allows a programmer to dynamically modify the size of blocks of memory that have been allocated, giving your software complete control over memory usage.

As an example, to allocate memory for one manager structure, use the following code (assuming our array of pointer declared eariler):

```
ptrMgr[0] = (struct managerStructure *)malloc (
    sizeof (struct managerStructure));
```

Remember that the size of the structure can be obtained using **sizeof**. For our manager structure, **sizeof (struct managerStructure)** will be converted to 69 by the C compiler.

Note the cast that was performed - **malloc** returns a generic pointer, hence, we must convert this to a pointer of type **struct managerStructure *** to keep the compiler happy, and once we've done this, we can assign this newly casted pointer into one of the 10 pointer entries we have available in the **ptrMgr** array!

**Useful Tip: sizeof**

You should **always** use **sizeof** in your software if you need to compute the size of your data objects in bytes, rather than counting the size by hand. Humans will make mistakes, compilers (technically) do not!

## 20.4. NULL

**malloc()**, **calloc()** and **realloc()** are not just limited to allocating dynamic storage for structures. For example, the following sample function will duplicate a string of **char**acters and return a pointer to the duplicated string. The function will check the incoming pointer for validity (check against NULL), and then allocate a block of memory the correct size (**strlen** of the incoming string, plus one for the new null terminator). If that is successful, a string copy is performed (**strcpy**), and then the new address of the string is returned. The function itself:

```
char * duplicate (char *s)
{
size_t len;
char *new;
```

```
    if (s == NULL)
    {
        return NULL;
    }
    len = strlen (s) + 1;  /* include null term too! */
    new = (char *)malloc (len);
    if (new == NULL)
        return NULL;
    strcpy (new, s);
    return new;
}       /* end duplicate */
```

In the function **duplicate()**, you may have noticed the use of a special constant **NULL**. C programmers have agreed upon a manner of recognizing when a memory allocation error has occurred. A special address marker called NULL is used to denote that memory could not be obtained. NULL is a predefined constant in **stdio.h**, and is generally set to the value 0 (but not always, hence it is safer to use NULL than zero in **all** programming you may do).

If there is no memory left in the system, you will find that **malloc()**, **realloc()** and **calloc()** will return the constant NULL. With this information, you have the ability to recognize when the system cannot give you any more free memory to allocate for your data, and you can then report this error to the user using your software, free up any unneeded memory that your program is using, or use disk techniques to offload the least used data structures to disk to free up more space.

Always check the return value of **malloc()**, **realloc()** and **calloc()** to see if memory was obtained or not. Using a NULL pointer in your software is simply inviting a crash of your software.

**Useful Tip: NULL Pointer Accessing**

It is interesting to note that advanced operating systems like UNIX and Windows will automatically terminate your software with a system error if your software ever assumes that a NULL pointer is a valid address. This is to protect all other applications that may be running concurrently, so that your inadvertent memory accesses do not corrupt any other programs. Typically, your software would continue to run, but cause unpredictable results, usually leading to a lockup of the system.

Remember, **always check** your addresses!

**Course Requirement: Check NULL**

Whenever a function can return NULL to indicate an error, check the return value.

## *20.5. free()*

When a program no longer needs dynamically allocated memory, it is customary (in fact, it is extremely good programming style) to return the unneeded memory blocks back to

the operating system. This is accomplished using the following function, prototyped in **stdlib.h**:

```
void free (void *ptr);
```

The free() **function** is used to tell the operating system that the memory pointed to by the generic pointer parameter **ptr** is no longer needed by your software, and this memory can be given to another program, or even your own program for a later allocation.

**Course Requirement: Freeing Memory**

Whenever allocating memory, always free up memory you do not require, after you have completed your memory operations. Do not assume that the operating system will do this for you.

A historical problem of this nature used to happen in the Windows 16 bit system – memory allocated for applications to hold images, icons or cursor data, if not manually deallocated by the programmer, were never freed up at all! This would cause memory to **leak**, and eventually, 16 bit Windows would die a miserable death. Again, programmers made the assumption that the operating system would do the cleanup, and unfortunately, Microsoft *forgot* a few areas to clean.

**ptr** is declared as a pointer to **void** since **free()** does not care what the pointer is pointing to, other than the fact that it must be a pointer to an already allocated block of memory. A program should always free up any allocated blocks of memory when about to exit. This leaves memory in its original state, and avoids wasting a precious operating system resource. Please note: **never** attempt to **free()** a statically allocated data object, only dynamic, as unpredictable results will occur!

## 20.6. Common Beginner Errors

- Always check the return value when you allocate memory.

## 20.7. Summary

In this chapter we had a look at the notion of **dynamic memory allocation**, which is used to request from the operating system memory only when the program really needs it. In this manner, memory is left free for other system needs such as being able to start up other programs as your software executes. This is how commercial software allows you to access the DOS Shell. To know how much memory to allocate using the functions **malloc()**, **realloc()** and **calloc()**, it is very advisable to use **sizeof()** to instruct the compiler to compute the number of bytes in the data object. It is wise to **free()** memory when it is no longer needed.

# Chapter 21. File Input and Output

## 21.1. Opening Notes

In this chapter, we will begin to explore the concepts behind creating and manipulating file-based data. The ability to save information into files, as well as being able to retrieve that information at a later date is of great interest to just about every programmer since applications like databases, spreadsheets, word processors and editors are rather useless without having a method of loading and saving the data that is manipulated.

## 21.2. Stream Based File I/O

We will begin our look at file operations by examining the concepts behind **stream based file input and output (I/O)**. The term "stream based" is a reflection on the type of information that will be going into and out of the files on disk: streams of characters, much like the character streams we place on the screen (character output) and character streams we receive from the keyboard (character input).

With stream based file I/O, what we are basically doing is placing our textual information that we would normally be sending to the screen, into a file instead. With stream based I/O, we can think of the file as simply another location our stream of output characters can be sent to. The reverse holds true as well: with stream based file input, the characters we would normally obtain from the keyboard will come from a file instead.

Please note that all functions and constants introduced in this chapter are prototyped in **stdio.h**.

## 21.3. Pre-declared Streams

Stream based file input and output has an inherent ability that every operating system supplies to us: not only can we send or retrieve streams from a file, we can send and receive information from other character oriented devices as well, such as the keyboard, the screen, a modem and the printer. In fact, every operating system has a number of built in character oriented streams that we can access, using the names:

- stdin: standard input
- stdout: standard output
- stderr: standard error output
- stdprn: standard printer input/output
- stdaux: standard auxiliary input/output

These values are useful when combined with the functions described in the next section below.

## 21.4. Standard File I/O Functions

With stream input and output, we use slightly different functions that are identical in nature to those used earlier, with the added ability of specifying where we want input/output to flow from/to. These functions are:

```
char *fgets (char *buffer, int length, FILE *stream);
char *fputs (char *string, FILE *stream);
int fprintf (FILE *stream, char *fmt, param1, ...);
int fscanf (FILE *stream, char *fmt, addr1, ...);
```

Notice that the functions appear almost identical to the standard input and output functions we have become used to, except for the additional parameter: a pointer to a special structure called **FILE**. **FILE** is a **typedef** for the streamed file structure, and is used to maintain specific information about the file that is being manipulated. The fields within this structure rarely concern a programmer, unless total control over file operations are desired, hence the structure members will not be discussed here.

The 5 standard streams that we saw earlier are in simply pre-declared pointers to file structures that every operating system has been designed to handle (ie: **stdout** is of type **FILE ***). Therefore we can print information to the screen using the function:

```
fprintf (stdout, "hello, how are you\n");
```
which is identical in execution to:
```
printf ("hello, how are you\n");
```

The use of the stream pointer allows us to specify exactly which file stream we want to work with. If we wanted to send a line of text to the printer, we would accomplish that feat using:

```
fprintf (stdprn, "this line goes to the printer\n");
```

By using different stream **FILE** pointers, we can send or accept information from different sources.

A discussion of these functions is not complete without a note on the capabilities of **fgets**. This function will accept a line of input from the specified stream. A line is a series of characters, terminated by a CR/LF pair (the carriage return and line feed pair of characters - generally, what is generated by hitting the Enter key on your keyboard), or the number of characters specified by the second parameter has been reached. The characters are read into the input buffer provided as the first parameter. This is a very useful replacement for the standard **gets** function, which has no upper limit on the number of characters read.

For example, let us assume that a programmer wished to read in a maximum of 50 characters into the array called **input**. The following code will perform this activity from the standard input (the keyboard):

```
char input[51];       /* remember to reserve 1 char
                       * for null terminator */

. . .

printf ("Enter some data (50 chars maximum): ");
fgets (input, 50, stdin);
```

**Useful Tip: sizeof and fgets()**

Use **sizeof** to get the second parameter to **fgets()**.

**Useful Tip: fgets() and '\n'**

It should be noted that **fgets()** will put a '\n' at the end of the string (but before the null-termination) if the maximum number of characters is not read in. To get rid of the '\n', you can use the following:

```
if( strlen(buf) > 0 )
{
    buf[strlen(buf)-1] = '\0';
}
```

A note about the **stderr** stream: in most operating systems today, the operating system allows the standard output and input to be **redirected** at the command line level. You may have seen commands such as:

```
DIR > LIST.TXT
```

which does a directory listing, and sends the listing into a file called LIST.TXT. No screen output will appear. The redirection always works on **stdin** and **stdout**. If your software wishes to display error messages on the screen even during times of redirection, the only method available is to use the **stderr** file stream, since it cannot be redirected (at least on most standard operating systems).

## 21.5. How to Open or Create a File

Useful as the above functions may seem, they are still limited by the fact that we haven't seen a manner in which we can create files on the disk our computer uses. The 5 standard streams above are connected to character based devices our computer is wired up to. There still must be a way of creating streams into a file on a disk, which is what we will look at now.

In C, to create a brand new file, or access an already existing file, we must call a function to initiate file-oriented operations. This all important function is called **fopen()**. The prototype for **fopen()** is:

```
FILE *fopen (char *filename, char *accessType);
```

Notice that calling **fopen()** will return a pointer to a stream based file. This returned pointer is then recorded and used by a programmer with the other functions we've seen above, in the same manner as the predefined character output streams. **fopen()** will return a value of NULL if there is any error in attempting to open the file stream to disk, such as trying to open a file on a non-existent disk, or non-existent subdirectory, etc.

## 21.6. Filename

The first parameter for **fopen()** is a pointer to a null terminated series of characters (string) which contains the filename we want to access. This filename must be a valid filename for the operating system we are working with. If we wanted to open a file called

TEST.DAT on drive C in the WORK subdirectory on a DOS or Windows system, the filename pointer would point to the string:

```
"C:\\WORK\\TEST.DAT"
```

Notice the use of the double back slash. Recall that since C compilers treat the back slash as a special character within strings (for the escape sequences like **\n** or **\a**), we must **escape** the backslash with a backslash. This is a problem on any machine that uses an operating system like Windows, since DOS uses the backslash to delimit subdirectory names. UNIX, on the other hand, uses the forward slash to delimit subdirectories, and there is no need to escape the forward slash.

Please note that the use of the double backslash is **only** needed for our source code in C on DOS and Windows machines - when accepting user input, the user need not type double back slashes, as user input is not escaped like C-based double quoted strings are.

## *21.7. Access Types For Opening Files*

The second parameter is a pointer to a **string** that specifies the access type of open operation we are performing. The valid access type strings include:

### Table 21-1: fopen() Access Types

| | |
|---|---|
| "r" | Open a file for reading only. |
| "w" | Open a file for writing only. If the filename does not exist on the disk, create a brand new file. If it does exist, erase the currently existing file so that we may start fresh. |
| "a" | Open a file for appending (writing) information to. If the file exists, allow new information to be added to the end of the current information, if not, create a brand new file. |
| "r+" | Open a file for reading **and** writing. Old information in a file can be overwritten with new information. |
| "w+" | Open a file for reading **and** writing. If the file does not exist, create it. Old information in a file can be overwritten with new information. |
| "a+" | Open a file for reading **and** appending. If the file does not exist, create it. |

Extra modifiers can be appended to the type string: **"t"** forces a text file to be worked with, **"b"** forces a binary file to be worked with. Binary files are discussed in the next chapter.

Notice that the **"w"** type string is lethal to files that already exist on the disk, and care must be taken to ensure that files that are important are not killed by the errant use of a **"w"** file open type. The **+** types are useful for situations where you need to read information and write new information over top of old information in a file, the definition of a random access file.

## 21.8. Closing an Opened File

Once a file has been opened, the programmer must ensure that the file is updated or terminated properly before leaving a program. This act is called **closing** a file (obviously the opposite of opening a file). When closing a file, the operating system ensures the completed file is intact on the disk, and that the disk directory information is updated to reflect the date and time of creation, and the proper file size.

To close a stream-based file in C, we use the following function, whose prototype is defined as:

```
int fclose (FILE *stream);
```

This function will accept a parameter of a pointer to a file stream (**FILE ***), and will inform the operating system that we are through manipulating the information stored within the file referenced by the file stream pointer. If any errors occur in closing, **fclose()** will return a non-zero value to inform you of such an error, otherwise zero returned indicates no error.

## 21.9. Putting It Together

Let's have a look at a sample program called **STREAM.C** which highlights some of the concepts we've seen in this chapter thus far.

```
/* STREAM.C */

#include <stdio.h>
#include <ctype.h>

int main (void)
{
char buffer[100] = {0};
int key = 0;
int x = 0;
FILE *filePtr = NULL;
    /*
     * ask user where char stream
     * output will flow to ...
     */

    printf ("where do you want output? ");
    printf ("\tF = file\n");
    printf ("\tP = printer\n");
    printf ("\tS = screen\n");
    while (1)
    {
        char input[10];

        gets (input);
        key = toupper (input[0]);
        if ((key == 'F') || (key == 'P') || (key == 'S'))
```

```
    {
        break;
    }
    printf ("try again!\n");
}/* end while */

/*
 * now get the appropriate stream pointer
 */

switch (key)
{
case 'F':
    /*
     * get a filename from user
     */

    printf ("enter a filename: ");
    gets (buffer);

    /*
     * create new file on disk
     */

    filePtr = fopen (buffer, "w");
    if (filePtr == NULL)
    {
        printf ("error in opening file!\n");
        return 1;
    }/* endif */
    break;


case 'P':// this case doesn't work under WinXP
    /*
     * use standard printer file pointer
     * (does not exist under WinXP)
     */

    filePtr = stdprn;
    break;

case 'S':
    /*
     * use standard output file pointer
     */

    filePtr = stdout;
    break;
```

```
    }/* end switch */

    /*
     * generate some character stream output
     */

    fprintf (filePtr, "this is text ");
    fprintf (filePtr, "for a stream\n\r");
    for (x = 0; x < 10; x++)
    {
        fprintf (filePtr, "%d times %d = %d\n\r",
                 x, x, x * x);
    }/* end for */
    fprintf (filePtr, "bye!\n\r");

    /*
     * if we created a file on disk,
     * we must close it before leaving
     */

    if (key == 'F')
    {
        if (fclose (filePtr) != 0)
        {
            printf("File cannot be closed\n");
        }
    }
    return 0;
}/* end main */
```

**Useful Tip: General program structure**

Here's an example of general program structure that commonly goes along with file processing:

1. Do stuff that happens at the end of the program.
    1.1 Declare variables and initialize them.
    1.2 Open files.
    1.3 Perform other one-time-only tasks.
    1.4 Sometimes, read the first record from the file.
2. Perform the main loop.
3. Do stuff that happens at the end of the program.
    3.1 Handle summary data.
    3.2 Finish up any output.
    3.3 Close files.

In this program, we ask the user to choose a destination for the character output. The choices are file, screen or printer. Once the user has chosen an appropriate destination, the program will obtain a valid file stream pointer for the output destination. For both the screen and printer, we set the variable **filePtr**, which is of type **FILE *** to **stdout** and **stdprn** respectively. We are taking advantage of the built in streams for these two output destinations.

For the file destination, we proceed to ask the user to enter a filename. Once entered, we call **fopen()** with a pointer to the buffer where the user had entered the filename, and a type of **"w"**, which will create a brand new file on disk (or overwrite an old file). If **fopen()** succeeded, **filePtr** will be set with a valid file stream pointer address, if not, an error message is issued to the user.

Once we have a valid file stream pointer, we can use it along with the file stream output functions. Here we've used **fprintf** to illustrate getting character based output into a streamed file.

Finally, if we had opened a file on disk, we must remember to close that file, lest we have the operating system improperly truncate the file. This is the purpose of the final **if** statement, where we check if the option chosen was to print the output to a file. If so, we will close the file, terminating the file correctly.

**Course Requirement: Closing files**

Most operating systems will automatically close opened files when the application terminates. This is **not** the most appropriate technique to use, for the same reasons as for dynamic memory. To ensure full portability of software across all operating systems, you must always manually close your files when no longer required, to avoid potential operating system deficiencies.

## 21.10. Common Beginner Errors

- Always check the return code from file-related functions. If there is an error, do something about it.
- Only use files that have been successfully opened.
- Watch out for the '\n' in fgets() buffers.

## 21.11. Summary

In this chapter, we have looked at the concepts behind character stream oriented file input and output. The term "character stream" refers to the fact that the information to be save in files is character, or ASCII oriented. We have seen that virtually all C compilers support five standard file input and output streams including the screen, keyboard, error display, printer and modem. A C programmer can use this fact to write very simple software to print information to a printer, or to extract characters from the serial port of the computer.

We also saw how to manipulate a file on a disk. The use of the **fopen()** and **fclose()** functions allow the programmer to create/open and terminate files whenever needed. A programmer should always close an opened file since (depending on the operating system) the operating system will not always do this job for you correctly.

# Chapter 22. Binary File Input and Output

## 22.1. Binary Files

In this chapter, we will build upon the concepts learned in the chapter on stream based file I/O and learn about the methods available in the C programming language to transfer **binary** information to and from a device or file.

The term "binary information" refers to the fact that the information is not in a textual, or human readable format. An executable program file in Windows is considered a binary information file since, if you took a brief look at the information stored within that file, you would not be able to make heads or tails of it. The binary information within such files is only useful for the task it was created for. An executable binary is useful only to the operating system (which allows such a file to be executed).

This also applies to information your own software may need to store. We've seen examples of data structures which contain a great deal of different types of information, some if it textual (character strings and buffers), but most in the form of double precision values, integers, long integers, and so on. These values are best stored in their natural state directly inside of a file, and binary file information transfer allows simple exchange to and from a disk of this type of information.

It should be noted that textual information, such as the stream file input/output we looked at in the last chapter, can also be considered binary information, only we just happen to be able to visually inspect the data stored within such streamed files.

## 22.2. Stream Based Binary File Techniques

We will begin to explore binary file I/O using variations on the functions we looked at for stream-based file I/O. It turns out that C allows a programmer to open files or devices in a streamed output format, and afterwards, read or write either streamed **or** binary information to or from such a file. Since we are already somewhat familiar with opening and closing of streamed files, we can look immediately at how to get binary information to and from such files.

As an example, we will use the following data structure to illustrate a binary block of data in the computer's memory:

```
typedef struct testData
{
    double balance;
    char name[20];
    short position;
    long socialInsuranceNumber;
} TESTDATA;

TESTDATA theData;
```

The **typedef** defines a structure called **TESTDATA** that contains four different data types: a long integer, an integer, an array of 20 characters, and a double precision floating

point value. The variable **theData** is a simple variable based upon the above structure. **theData** occupies 34 bytes of memory.

At this juncture, we know how to place the character string into a file. Simply use the following:

```
fprintf (outFile, "%s\n", theData.name);
```

and the **name** field of the variable **theData** will be sent into the file pointed to by **outFile**. We are of course assuming that a file was correctly opened using **fopen()** earlier to allow **outFile** to point to a valid **FILE** structure.

Now suppose a programmer wishes to save not only the **name** field in the file, but the **balance**, **socialInsuranceNumber** and **position** fields as well. Currently, the only method at our disposal for sending that information to a file is to perform a number of **fprintf()** calls and format the information in a manner that a **fscanf()** can retrieve the information later. Although such a scheme may work, it is exceedingly inefficient, since a file on just about any operating system in use today has no more problem storing the bytes which make up a double precision floating point value than the bytes storing the **name** field. The operating system could care less if the information it is to save on disk is human readable or not.

This is where binary file I/O comes into play. Firstly, the file must be opened using **fopen()** in a binary format. This is done by appending a **b** after the letter which represents the file-opening mode. To read a binary file, simply use a type string of **"rb"** or **"r+b"**, and to create a binary file, use **"wb"** or **"w+b"**. Next, we need to somehow inform the compiler we wish to transfer binary information. The following functions are employed in the ANSI C standard library of functions (prototyped in **stdio.h**) to place or retrieve complete blocks of information to and from a streamed file or device:

## 22.3. fread() and fwrite()

The following prototypes are instrumental in working with binary data:

```
unsigned int fread (void *ptr, unsigned int size,
         unsigned int nitems, FILE *stream);

unsigned int fwrite (void *ptr, unsigned int size,
         unsigned int nitems, FILE *stream);
```

Both **fread()** and **fwrite()** operate on blocks of information, rather than character based strings. Notice that for both functions, as was the case for **free()**, the first parameter is declared to be a pointer to **void**. This simply means that they will accept **any** type of pointer, with the keyword **void** being a convenient method of saying "pointer to anything".

The second parameter of both functions is an unsigned integer that specifies the size of a single unit of the data block to transfer. With an unsigned integer, the maximum block unit size, in 16 bit DOS, is 65535 bytes (64K in total), or in a modern 32 bit operating system, over 4 billion bytes in total.

The third field is another unsigned integer that specifies the total number of units to be transferred. The maximum number of bytes to be transferred at one time by either of these calls is **size * nitems** which amounts to a huge number of bytes!! Although capable of massive amounts of information input/output, typically these functions are called to transfer only one block at a time, or one structure array at a time.

The final field is, of course, a pointer to a **FILE** structure, which specifies the stream we are writing to. Remember, this stream must be pointing to a binary file, opened by appending 'b' to the standard type string for **fopen**

The return value of these functions indicates the total number of **complete** units that were transferred using these functions. If a value less than the number of units specified is returned, it is generally assumed an error occurred during transfer. This error could be something like end of file, not enough room on disk, and so on.

When performing error checking on our file output, the return value of these functions is a convenient value to check - if the returned number of blocks accessed does not match the number of blocks requested, then report an error. Otherwise, assume all is well! What could be simpler than that!

With our structure we defined earlier, we could save away the contents of the variable **theData** by using the call:

```
fwrite (&theData, sizeof (TESTDATA), 1, outFile);
```

We can read the information back into main memory by using:

```
fread (&theData, sizeof (TESTDATA), 1, outFile);
```

To transfer blocks of information to and from disk it is simply a matter of calling **fread()** and **fwrite()** with the appropriate block size and unit numbers. These functions are as simple to use as the character stream functions, with the added ability of being able to read and write binary blocks.

**Useful Tip: Use sizeof with fread() and fwrite()**

To get the sizes for use with fread() and fwrite(), use sizeof.

## 22.4. Putting It Together

The following is a sample program which uses all of the file input/output functions we've seen so far:

```
/* fwrite.c */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

typedef struct
{

    double balance;
    char name[20];
```

```
    int position;
    long socialInsuranceNumber;
} TESTDATA;

int main (void)
{
char buffer[100] = {0};
FILE *fp = NULL;
TESTDATA theData;

    /*
     * obtain information from user
     */

    printf ("Enter name: ");
    gets (theData.name);
    printf ("Enter balance: ");
    gets (buffer);
    theData.balance = atof (buffer);
    printf ("Enter position (0 to 9): ");
    gets (buffer);
    theData.position = atoi (buffer);
    printf ("Enter S.I.N.: ");
    gets (buffer);
    theData.socialInsuranceNumber = atol (buffer);

    /*
     * save information to disk
     */

    fp = fopen ("test.dat", "wb");
    if (fp == NULL)
    {
        printf ("Error: can't open for writing\n");
        return 1;
    }/* endif */
    fwrite (&theData, sizeof (TESTDATA), 1, fp);
    if (fclose (fp) != 0 )
    {
        printf("Cannot close file\n");
        return 1;
    }

    /*
     * now reset the fields, and
     * read the block back in
     */
```

```
    theData.name[0] = 0;
    theData.position = 0;
    theData.balance = 0.0;
    theData.socialInsuranceNumber = 0L;

    fp = fopen ("test.dat", "rb");
    if (fp == NULL)
    {
        printf ("Error: can't open for reading\n");
        return 2;
    }/* endif */

    fread (&theData, sizeof (TESTDATA), 1, fp);
    if (fclose (fp) != 0 )
    {
        printf("Cannot close file\n");
        return 2;
    }

    /*
     * print data to screen
     */

    printf ("Name: %s\n", theData.name);
    printf ("balance: %f  position: %d SIN: %ld\n",
    theData.balance, theData.position,
    theData.socialInsuranceNumber);
    return 0;
}/* end main */
```

## 22.5. Other Stream Based Functions

A few other functions to look include:

```
int fseek (FILE *stream, long offset, int whence);
```

The **fseek()** function allows a programmer to move about within a streamed file using the **offset** parameter to specify how many bytes to move. The parameter **whence** is set to one of the following:

*SEEK_SET (from the file beginning)*
*SEEK_CUR (from the current position in file)*
*SEEK_END (file end)*

With this function, movement relative to start, current or end of file is possible, allowing any file traversal to be accomplished. A value of zero returned specifies that no major errors have occurred. A note of warning: depending on your compiler, the implementation of **fseek()** will not always return an error if the end of file is breached, thus take care when using this function.

```
void rewind (FILE *fp);
```

This function will reset the file position back to the start of the file on disk. This is identical to **fseek (fp, 0L, SEEK_SET);**

```
long ftell (FILE *fp);
```

This function will return the current position within a file, or -1 on error. It is useful for recording a specific location in a file, in order to be able to return to it later. In the following example, a function called **fileSize()** shows how you can determine the size of a file:

```
long
fileSize (char *filename)
{
    long pos = 0L;
    FILE *fp = NULL;

    if ((fp = fopen (filename, "r")) == NULL)
        return -1L;

    fseek (fp, 0L, SEEK_END);/* to end of file */
    pos = ftell (fp);
    fclose (fp);
    return pos;
}/* end fileSize */
```

Here we are opening a file, and we will return -1L if an error has occurred. If successful, we reposition, using **fseek()** to the end of the file, read the position using **ftell()**, and close the file. We finally return the size to the caller.

```
int feof (FILE *fp);
```

This function can inform you through a non-zero return value if the end of a file has been breached. Note that you must actually pass the end of the file before **feof()** will inform you.

## 22.6. Common Beginner Errors

- Always check the return codes from file-related functions.
- Make sure that you distinguish between text and binary files.

## 22.7. Summary

In this chapter we explored the concepts behind binary file input and output. Binary is a term used to describe data in its natural form. With binary file transfer, a programmer is able to extract the data within integers, double precision floating point values, or even structures and unions with virtually no effort whatsoever.

We looked at the use of streamed binary file transfer methods, using the functions **fread()** and **fwrite()**. These functions operate on files that have been opened using **fopen()**. These functions allow transfer of blocks of data to and from disk, amounting to large amounts of file transfer with very little work on the part of the programmer.

# Chapter 23. Parameters To main()

## 23.1. Introductory Notes

Another interesting feature of the C language is the command-line arguments that are passed to the **main()** function. In reality, DOS, UNIX (or any operating system for that matter) will always supply three parameters for your **main()** to use, if it chooses.

## 23.2. main() Parameter Example

As an example, study the following code fragment:

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

The three parameters specified for **main()** tell the **main()** function everything it needs to know about what the user typed on the command-line to start your program. This comes in useful for something like the TYPE command in DOS or "cat" command in UNIX: these applications must have one argument, the filename to display on the screen. Hence, the command must know how many **arguments** have been supplied, and what the actual arguments are.

**argc** is an integer which tells you how many arguments are on the command-line. Normally, with no arguments, **argc** is 1, meaning that the only thing the user typed is the name of the program itself. The second parameter, **argv**, which is defined as an array of character pointers, will have element 0 set to the address of a string which is the name of the program. Other parameters will be set in elements 1, 2, etc of this array. If you had a program called FOO, and you typed on the DOS command-line:

```
C:\> FOO HELLO HOW ARE YOU
```

argc will be set to 5, and **argv[]** will be set to:

```
argv[0] = a pointer to FOO
argv[1] = a pointer to HELLO
argv[2] = a pointer to HOW
argv[3] = a pointer to ARE
argv[4] = a pointer to YOU
```

The spaces between arguments on the command-line delineate the individual arguments, which in turn tells the operating system how many character pointers to give to the **main()** function.

**env** is available from most operating systems, and is a list of **environment strings** that are present in your computer's memory. These strings inform various programs about configuration information. For example, a common environment variable is:

```
PATH=C:\;C:\DOS;C:\WINDOWS;C:\GAMES
```

and tells DOS where to locate executable programs. You have access to these strings through the character pointer array **env**. The last pointer in the array will have its value set to 0 (usually called a **NULL** pointer).

Because both **env** and **argv** are arrays of character pointers, you can use these character pointers in any software that works with strings. For example, to output argument number two in the array **argv** as a string on the display, use:

```
printf  (“argv at offset 2 is: %s\n”, argv[2]);
```

The %s formatting code can be used in a **printf()** call to output an **argv** array member or an **env** array member.

**Useful Tip: argv[] Validity**

Never access argv elements without first checking argc to make sure that the argv array index is valid.

## 23.3. Common Beginner Errors

- Always check argc to see if argv's array index is valid.

## 23.4. Summary

main() can get information from the system through its parameters. argc and argv are useful for getting arguments from the command line. env can provide environment variables as strings.

# Chapter 24. Operating System Functionality

## 24.1. Introductory Notes

In this chapter, we will explore how to work with some the functionality that your operating system gives you, in terms of disk and other low level resource access, deep within the bowels of your operating system.

An operating system (OS) is a special program that generally needs to be loaded first, before any application software is loaded. The OS will provide rudimentary access to data on your system, by supplying command-line or point and click access to files, notes, word processing documents, and so on. The OS is responsible for maintaining this data on long-term storage.

The operating systems used today are incredibly powerful compared to those of only a decade ago. High-level systems like UNIX have migrated down to the level of desktop users, through Linux. Windows XP and Vista are now de-facto standards for window-based (graphical) systems. Smaller operating systems such as QNX or Embedded Linux are often used in "real time systems" where access to incoming data is crucial and time-sensitive. Note that there is no one OS that is suitable for all tasks, as each have their strength or weaknesses.

In relation to files, your OS is responsible, when working with disks, of determining what type of media is being used, the size of the disk, the capacity of the disk, the number of heads, cylinders, tracks, and so forth, without ever losing a single byte of information a user has requested to be saved. Because of these rather enormous responsibilities, the makers of the various stable versions of your OS ensure that all new types of disks are taken into account so that no matter what type of hardware is connected to your PC, there is a tested and proven method of getting information to and from that hardware.

As a programmer in C, you will find that occasionally, there is a need to get closer to the disk than the file I/O functions we've seen so far. For instance, there may be times you will want to display to the user the files in a certain directory, so the user of your software can see at a glance what filenames have already been chosen. Of course the user can just blindly type the name in, and hope that a file doesn't already exist with that name, but it is always more user friendly to show as much information to the user as possible to help that user make informed decisions when working with your software.

With this chapter, we will take a look at some of the more useful functions that operating systems supply to a C programmer so that we can get information such as the above with very little effort on the programmer's part.

## 24.2. Listing of Files

In most OSes, there is a commonly used command to list files. In DOS, it is DIR. In Linux, it is ls. DIR or ls is used to ask the OS command interpreter (the software that understands what you are asking for) to list all of the files in a certain location on disk. The OS will take the options for the listing command (which are called **arguments**), and

begin to search the required area of the disk to see what files exist there. DIR will then report to you pertinent information about all of the files that it finds.

With the listing command, you should be aware that an OS will accept **wildcard** options. For instance, on a DOS system, if you wanted to find all C files within the directory called WORK on drive C:, you would enter the command:

```
DIR C:\WORK\*.C
```

A similar capacity exists on a UNIX system using the generalized ls command:

```
ls /work/*.c
```

The OS will search that area of the disk, and list whatever files exist that end with an extension of **.C**.

To achieve this same result in our software, we must ask the OS to supply to our software a listing of files in the desired location on a disk.

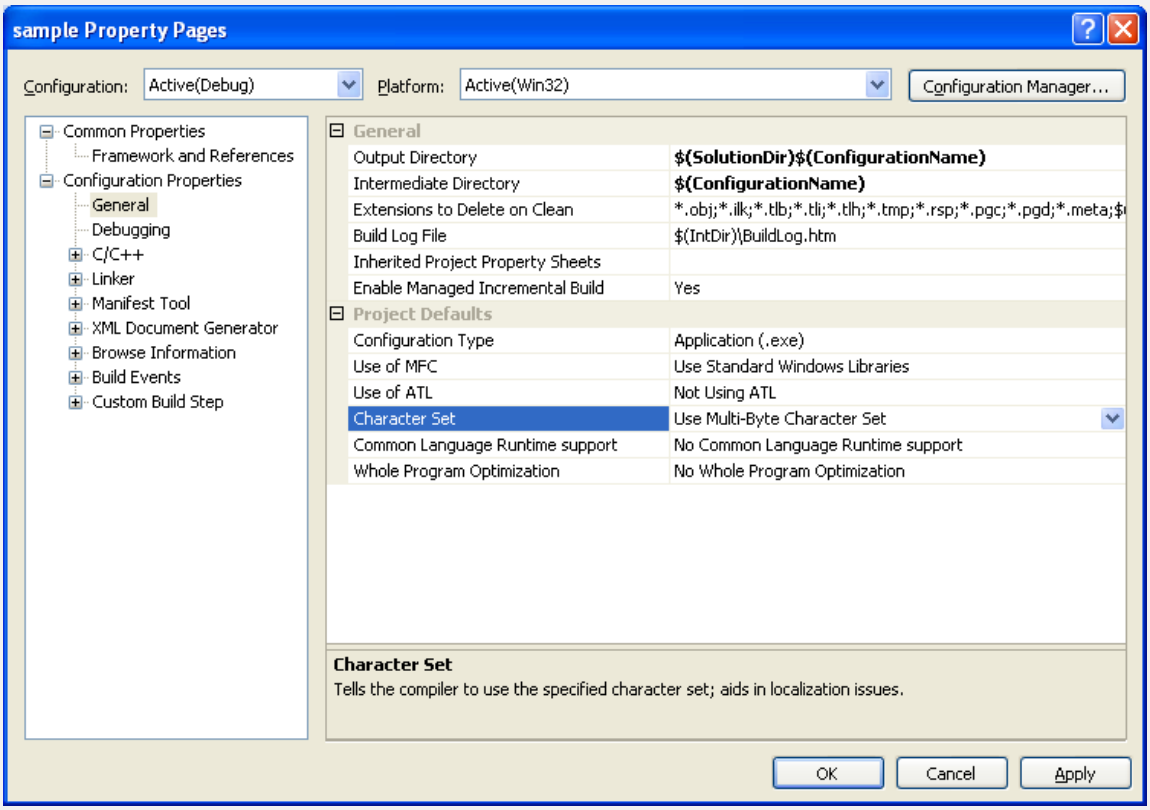
## 24.3. Windows 32-bit Search Functions

For 32-bit Windows programs, both graphical and command-line based applications can use the following Windows-specific functions:

```
#include <windows.h>

HANDLE FindFirstFile (char *filename, WIN32_FIND_DATA *w);
BOOL FindNextFile (HANDLE h, WIN32_FIND_DATA *w);
BOOL FindClose (HANDLE h);

/* example */

WIN32_FIND_DATA fd;
HANDLE h;

. . .

h = FindFirstFile ("*.C", &fd);
if (h != NULL)
{
    do
    {
        printf ("Name: %s, Size: %ld\n",
        fd.cFileName, fd.nFileSizeLow);
    } while (FindNextFile (h, &fd);
    FindClose (h);
}       /* endif */
```

**Useful Tip: Compiling Projects involving FindFirstFile()**

In order to get FindFirstFile() to work properly in Visual Studio, you **must** change the default character set for your Project to be Multi-Byte instead of Unicode. This is done under the Project Properties.




## 24.4. Changing Directories

When information must be searched for on a particular location on a disk, but the position differs from installation to installation of the software, a programmer may want first to switch to that particular location prior to doing any directory searches. This can be accomplished by telling the operating system to change to the desired working directory. On Windows, this is accomplished through the use of the **_chdir()** function:

```
#include <direct.h>

int _chdir (char *path);
```

The desired directory path to change to is passed as a parameter. The function will return -1 if an error has occurred.

## 24.5. Renaming a File

Often a programmer desires to change the name of a file. Such a situation occurs most frequently when manipulating an important file. For instance, if a database is to generate a report, the database program will usually try to rename any existing report file to a backup file, before writing the new report. With this action, the database can output the desired report, while keeping a backup of the old file still on disk.

In C, we can rename a file using the following function:

```
#include <stdio.h>

int rename (char *oldname, char *newname);
```

This ANSI C function will return 0 if successful, -1 if unsuccessful. The two parameters are simply pointers to the old name of the file and the new name of the file. An error will occur if **rename()** discovers that the original file does not exist, or if the new file name already exists on the disk, or if the new name is invalid. An example of this command is illustrated in the following program, RENAMEIT.C. (Note: we cannot give this program the name RENAME.C since the executable will then have the same name as the DOS internal command RENAME!)

```
/* RENAMEIT.C */

#include <stdio.h>

int main (int argc, char *argv[])
{
    if (argv != 3)
    {
        printf ("Error! Need old and new names.\n");
        return 1;
    }/* endif */
    if (rename (argv[1], argv[2]) != 0)
    {
        printf ("Error during rename.\n");
        return 2;
    }/* endif */

    return 0;
}    /* end main */
```

The sample above will use the **rename()** function to rename the file specified as the 2nd argument on the command line (remember, the name of the program is always the 1st argument on the command line), and to the file specified as the 3rd argument. Note that a rename is also a file mover - to move the file to another directory on the same drive is no different than just supplying a new name! Give it a try and see in action!

## 24.6. Deleting a File

Along with the above operation, a programmer will often desire to delete a file from disk. This is done using the following ANSI C function:

```
#include <stdio.h>

int remove (char *filename);
```

It too returns 0 if no errors occurred, -1 if an error exists.

## 24.7. The system() Call

If you want to make use of an operating system facility that is not readily available through a function call, you can use the system() function. system() is compatible with all operating systems, hence, it is valuable for portability.

```
#include <stdlib.h>

int system (char *command);
```

This function will pass the string pointed to by **command** to a copy of the command interpreter available on your machine and the command interpreter will in turn execute this program.

As an example:

```
/* SYSTEM.C */

#include <stdio.h>
#include <stdlib.h>

int main (void)
{
        system ("dir *.c");
        system ("mem");
        return 0;
}    /* end main */
```

It should be noted that the use of the system() function call is generally frowned upon because it is very resource-intensive (it starts a completely new command interpreter).

## 24.8. Common Beginner Errors

- Make sure that the call that you are using will work outside of your IDE and your particular computer.

## 24.9. Summary

In this chapter, we explored the topic of extracting low-level information from the operating system of a computer. We studied how to obtain information such as file listings from the Windows operating system so that a program can inform a user that certain files already exist on disk. Such information is very valuable to a user of a

program since it enables the user to make informed choices when it comes to saving files of data.

We also explored the ability of a C programmer to change working directories and drives, rename and delete files from disk. Again, offering options in a software package such as these allow much flexibility to the user of a program.

Finally, we took a look at the system() function to perform actions that aren't readily available through other means.

# Chapter 25. Recursion

There are times in programming when it becomes necessary to call a function from inside the function that is currently executing. Sometimes when a function is called, it calls a second function, which in turn will call the original function. These situations are usually referred to as **recursion**, and the act of recursion is simply the act of calling oneself a second time before the original call has had a chance to complete.

It may appear that the above idea is horribly complicated, and rather stupid as well! There are actually a number of advanced algorithms which benefit from this idea of recursion to efficiently perform a task.

## *25.1. Recursion Example*

The following program, STARS.C, utilizes recursion to output a certain number of asterisks to the screen. Note, only a single **if** statement is used to control the sequence of events in this example. /*
 * STARS.C
 */

```
#include <stdio.h>

void stars (int n)
{
    printf ("*");
    if (n > 1)
    {
        stars (n - 1);/* recursion! */
    }
}/* end stars */

int main (void)
{
    printf ("printing 5 stars ... ");
    stars (5);
    return 0;
}/* end main */
```

In the **main()** function above, we are calling the function **stars()** with an integer parameter of the constant 5. We are telling **stars()** that we desire 5 stars to be output to the screen. **stars()** begins naturally enough, printing a single star.

The **if** statement that follows instructs the compiler to generate code to check if the current value of the parameter **n** is greater than 1. If so, another call to **stars()** is made, except this time we tell **stars()** to print one less star!

Notice that we have called **stars()** a second time, before we've finished executing **stars()** the first time. This is where the recursion takes place. In fact, **stars()** will continue to be

called, each time receiving a parameter one less than the previous call, until the parameter **n** is equal to 1. Once this point is reached, the function terminates normally, and all previous functions will too terminate as coded. This is how a recursive function call completes - it "unrolls" itself back to the original caller.

Although this is a rather trivial example of a recursive algorithm, it nevertheless demonstrates how this concept works. In real life, we are exposed to recursive situations daily. Take, for instance, the task of locating a file on a DOS hard disk. You will perform DIR commands, either locating a file, or locating a potential sub-directory where a file may be stored. You will then switch into that sub-directory, and do another DIR. If you find another sub-directory, you can switch there too. Once you've exhausted a search in a particular directory, you must switch back to the previous level. This is exactly how the **stars()** function performed its recursion.

The main disadvantage of recursion is that if it is not managed correctly, eventually, memory resources within your program will be exhausted. The CPU has only a limited amount of space where it can keep track of parameters to a function, and by recursively calling a function, we slowly chip away at this memory. For this reason, most programmers steer clear of recursion unless it is absolutely necessary, since other flow of control statements can suffice in most cases, and are easier to maintain as well.

## 25.2. Recursion vs. Looping

It should be noted that the real-life uses of recursion are quite limited. When learning about programming for the first time, it is tempting to use recursion instead of implementing a simple loop. Resist the temptation. Loops are almost always superior, due to their simplicity and clarity. In other courses, you will learn about situations in which recursive algorithms are desirable (e.g. traversal of binary trees). Wait until then.

**Useful Tip: Recursion Usage**

Don't use recursion unless you are absolutely certain that it's the best way to do the job (or unless you're told to do so in an assignment ☺).

## 25.3. Common Beginner Errors

- Always make sure that the recursion can end.

## 25.4. Summary

In this chapter, we learned about calling a function from within itself (or another function called by itself, etc.). Recursion has a place in programming but it should not be used to replace proper design using loops.

# Chapter 26. Advanced Pointer Usage

## 26.1. Pointer Review

In this chapter, we will look at some advanced programming techniques that utilize and expand our knowledge of pointers.

As you recall, a pointer variable is simply a piece of memory sized large enough to hold an address for the computer system we are working with. The pointer variable is an address variable. The data type associated with a C pointer variable declaration is just a reference to the type of data the address points to. For instance, a pointer to a **char** is an address of where in our computer's memory a character (or byte) of memory may exist. The pointer allows us to access memory in the form of the data type the pointer variable is declared as. A **double** pointer allows a program to view memory in chunks of 8 bytes, since a double precision floating point data object is 8 bytes in size.

It should be noted that a common term used to describe the act of referencing an object in memory via an address is **indirection**.

## 26.2. Pointers to Pointers

In C, it is possible to declare that a pointer variable will contain an address of where another pointer exists in memory. Since a pointer variable is used to hold an address, it should be quite apparent by now that an address can point to anything, so why not another pointer?

The syntax for declaring a pointer to a pointer is as follows:

```
data-type **name-of-variable;
```

Notice in the syntax the use of ******. The double asterisk is used to inform the compiler that what we have is a pointer to a pointer. The single asterisk is used to declare a pointer to a data object, thus two asterisks in the declaration declares a pointer to a pointer to a data object.

Some examples of valid pointer-to-pointer declarations (without initializations) are:

```
int **x;
double **ptr;
char **argv;
```

The first example declares **x** as a pointer to a pointer to an integer. The second example declares that **ptr** is a pointer to a pointer to a double in memory. It should be no surprise then that **argv** as defined above is a pointer to a pointer to a character in memory.

To access the data that a pointer to a pointer points to, we must use **double indirection**. We must first use the contents-of operator to view what is in memory that the variable itself points to. Then, we must use this address to peek at what is in memory that this second address points to. To assign the letter **'A'** into the character using the pointer that **argv** is pointing to, use:

```
**argv = 'A';
```

Notice that in our code, we again use the asterisk, except, as for normal pointer operations, the asterisk is now the contents-of operator. Using two of these operators tells the compiler that what we want is the "contents of the contents of argv".

Study the following scenario:

```
argv contains address 10124.
In memory at address 10124 is the address 1779.
In memory starting at address 1779 is the character array
    "hello".
```

By referencing the pointer **argv**, we get the value 10124, and we can access the address of the character pointer that **argv** points to with the contents-of operator. Hence, ***argv** will then extract a "pointer to a character" data object, which in this case is the address 1779. If we want to look in memory at what this character pointer points to, we must use another contents of operator. This will obtain the actual character 'h', since this is the **char** that ****argv** is pointing to.

The above scenario is seemingly idiotic. It is not readily apparent why one would be interested in using a pointer to a pointer. One example is where a function is supposed to return three pointers to data objects in memory. Since a function can normally only return one piece of data at one time, we can use pointers to pass the addresses of pieces of data to a function in order for that function to fill in the appropriate information. If a function is supposed to generate three distinct addresses at one time, then we must pass to this function the addresses of three pointers - we must declare three pointers to pointers!

What if you wanted a function to return the addresses of where the drive letter, the start of the subdirectory path, and the start of the filename exist inside of a full DOS filename specification. A possible prototype for this function could be:

```
void locatePathInfo (char *fullPath, char **drive, char
     **path, char **name);
```

Notice in our ficticious prototype, the first parameter is a normal pointer to a **char**. We can pass to this function a string of chars such as:

```
"C:\\WORK\\PRINTER\\EXAMPLE.OUT"
```

The next three parameters are declared as pointers to pointers. A segment of C code which calls this function can be written as:

```
char *d = NULL, *p = NULL, *n = NULL;
. . .
locatePathInfo ("C:\\WORK\\PRINTER\\EXAMPLE.OUT",
     &d, &p, &n);
```

This function, when finished, will have the character pointer **d** set to point to the drive letter C in the full path, the pointer **p** is set to point to the letter W of the path specification, and the pointer **n** is set to point to the letter E of the filename EXAMPLE.

Another situation where pointers to pointers are used is the fact that an array of pointers can be thought of as a pointer to a pointer. Study the following declaration once again:

```
char **argv = NULL;
```

**argv** is a pointer to a pointer to a character. Because we can use array indexing on pointer variables, then address information can be located and read out using offset notation. For example:

```
argv[3]
```

will be the 4th **char *** in memory from the base address of that array. We can use a pointer to a pointer as an array of pointers in memory. In fact, many programmers will declare the **main** function in two standard methods:

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

or

```
int main (int argc, char **argv, char **env);
```

It should be apparent that both declarations are equivalent since a pointer to a pointer is nothing more than an alternate declaration of an array of pointers.

## 26.3. Pointers and Functions

In C, we have seen that the name of an array is a pointer to the first element in the array. It turns out that the name of a function is also a pointer: it locates the first byte of code for that function in memory! In fact, the rounded brackets after a function name are an operator in C that informs the compiler that we are dealing with function instead of a variable.

Again, a question will arise: what purpose does a pointer to code serve?

Sometimes a function will need to be told of another function it will call indirectly. This can be accomplished by passing it the address of another function in memory.

One such common example is the ANSI C standard library function **qsort**. This function is prototyped as:

```
void qsort (void *base, unsigned int num, unsigned int
     size, int (*compare)(const void *, const void*));
```

This is probably one of the most complicated prototypes we've encountered so far. Notice the use of the **void *** data type. The **qsort** function has only one purpose: to sort objects in memory. These objects must be consecutively stored (an array of objects), thus, by declaring the pointers as pointers to **void**, we can allow **qsort** to work with any data type in C, as **void *** is nothing more than a pointer to anything. The second and third parameters inform **qsort** of how many objects are consecutively in memory, and the size in bytes of each data object.

Study the last parameter in the prototype. This is a very strangely declared parameter, which informs the compiler that the last parameter is a **pointer to a function**. The declarations is:

```
int (*compare)(const void *, const void *)
```

and declares a parameter variable called **compare** as a pointer to a **function** that requires two **const void *** parameters, and will return an **int** data object. This pointer is required for the **qsort** function in order for **qsort** to know how to sort the data in the array! By

indirectly calling this second function through the function pointer that is passed, **qsort** can easily sort any data object array because the programmer specifies an address of a function that will do the comparisons required for sorting!

To pick this declaration apart in more detail, **compare** is declared as a pointer, because of the asterisk in front of the name **compare**. So far we have the start of a generic pointer declaration. But notice that we enclose the declaration inside of a pair of round brackets. This is what makes this a declaration of a variable that contains the address of a piece of code, as distinguished from a data pointer, that will not have the brackets around the declaration. We follow this with another pair of round brackets with prototyped parameters. This lets the compiler know what the parameter list of the function is like, so that type checking can be done at compile time.

Because it is a function, functions must always be prototyped to describe not only input parameters, but also the output data type. The **int** declares that **compare** is a pointer to a function that will return an **int**.

## 26.4. Pointer to a Function Notation

The standard notation for declaring a pointer to a function is:

```
return-data-type (*name-of-pointer)(parameter
    declarations);
```

As ANSI C is becoming more and more popular (and because C++ demands the use of prototypes), it is suggested that all functions and pointers to function be declared with the data types of the parameters.

As a short example of declaring a pointer to a function, a pointer to a function, called **foo**, will be used to contain the address of another function that will return a character pointer, and requires two character pointers as input parameters. The declaration is:

```
char *(*foo)(char *a, char *b);
```

Ensure you can follow the above declaration, as this understanding is crucial to your ability to work with function pointers later in your programming.

## 26.5. Putting It Together

Another example will quickly clear up confusion about this function, and the use of a pointer to a pointer.

```
/* QSORT.C */

#include <stdio.h>
#include <stdlib.h>

int up (const void *x, const void *y);
int down (const void *x, const void *y);
void dumpArray (int *a, int num);

int up (const void *x, const void *y)
```

```
{
int a = 0;
int b = 0;

    a = *(int *)x;
    b = *(int *)y;
    if (b > a)
    {
        return -1;
    }
    else if (b < a)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}/* end up */


int down (const void *x, const void *y)
{
int a = 0;
int b = 0;


    a = *(int *)x;
    b = *(int *)y;
    if (b > a)
    {
        return 1;
    }
    else if (b < a)
    {
        return -1;
    }
    else
    {
        return 0;
    }
}/* end up */


void dumpArray (int *a, int num)
{
int x = 0;

    for (x = 0; x < num; x++)
    {
```

```
        printf ("%3d ", a[x]);
        if ((x % 10) == 9)
        {
            printf ("\n");
        }
    }/* end for */
}/* end dumpArray */

int main (void)
{
int a[20] =
{
    19, 99, 5, 10, 7,
    33, 22, 57, 50, 48,
    3, 77, 66, 13, 18,
    9, 10, 59, 38, 28
};


    printf ("This program will quick\n");
    printf ("sort the array a\n\n");

    printf ("original contents:\n\n");
    dumpArray (a, 20);

    qsort (a, 20, sizeof (a[0]), up);
    printf ("\n\nsorted upwards:\n\n");
    dumpArray (a, 20);

    qsort (a, 20, sizeof (a[0]), down);
    printf ("\n\nsorted downwards:\n\n");
    dumpArray (a, 20);
    return 0;
}/* end main */
```

The program QSORT.C demonstrates how to use the **qsort** function and the use of pointers to functions. Note in the **main** function how we call **qsort**. The first call is passing to **qsort** the address of the data to sort (the name of the array **a**), the number of elements (20), the size of each element (**sizeof (a[0])**, which is requesting the compiler to compute the size of element 0 of the array, which will be the size of a single integer), and finally, the pointer to the function that will perform the comparisons. In this case, the function is called **up**, and is defined earlier in this program.

Notice that to pass the address of the function, we simply use the name of the function, much like how we used the name of the array **a** to pass a pointer to the first element to sort. At this point, **qsort** knows enough information about this array and its data objects to be able to efficiently sort this data. The qsort is short for "quick sort", and is a very efficient sorting algorithm.

Check your compiler's help documentation to determine how to write a sorting comparison function, similar to **up()** and **down()** in the QSORT.C program. You will find that **qsort()** requires a specific format to the return values for the comparison functions, which is why **up()** and **down()** are designed in such a curious manner.

## 26.6. Arrays of Function Pointers

There are times where a programmer may want to call a function based upon the current state of a variable. At the moment, the most efficient manner to do such a thing would be to use a **switch** statement, comparing the value of a variable against a number of constant states, and executing a function as a result of a successful comparison.

Unfortunately, if there are a large number of states to compare against, a large number of time consuming comparisons may be done. As a state further and further down the list of cases is compared, the longer it takes to complete the **switch** statement. In such situations, especially where program performance is crucial, a programmer may resort to defining an array of function pointers rather than working with a large switch statement.

This is a programming idea borrowed heavily from software such as operating systems. In order for an operating system interrupt to be dispatched as efficiently as possible, the operating system will define a **table**, or array of function pointers to the operating system functions, and uses the function number that has been requested as an index into the table. Once the address of a function is located in the table, it can simply jump to that address by calling the function obtained as a pointer from the table. This avoids time consuming **switch** comparisons and speeds up operating system performance without much extra coding effort.

## 26.7. Putting It Together

Another small example program will highlight how a table of function pointer can be setup and used:

```
/* PTRFUNC.C */

#include <stdio.h>
#include <stdlib.h>

/*
 * function prototypes
 */

int func1 (int);
int func2 (int);
int func3 (int);
int func4 (int);
int func5 (int);
/*
 * notice the "bizarre" declaration for the array
 * ptrsToFn
 *
```

```
 * its data type is:
 *
 * int (* )(int)
 *
 * meaning: a pointer to a function that requires an
 *          integer parameter and will return an integer
 *          each element still
 *          only takes up 1 integer's amount of room in
 *          bytes of memory!
 */

int (*ptrsToFn[5])(int) =
{
    func1,
    func2,
    func3,
    func4,
    func5
};

/*
 * declaration of a simple pointer to a function
 */

int (*otherPtr)(int) = func3;

/*
 * the actual functions themselves
 */

int func1 (int a)
{
    printf ("this is function 1\n");
    return (1);
}


int func2 (int a)
{
    printf ("this is function 2\n");
    return (a * 2);
}

int func3 (int a)
{
    printf ("this is function 3\n");
    return (a * 3);
}
```

```
int func4 (int a)
{
    printf ("this is function 4\n");
    return (a * 4);
}

int func5 (int a)
{
    printf ("this is function 5\n");
    return (a * 5);
}

/*
 * let's demo this stuff ...
 */

int main (void)
{
int x = 0;
int y = 0;

    printf ("A demo of pointers to functions ...\n");

    y = 0;
    for (x = 0; x < 5; x++)
    {
        y = (ptrsToFn[x]) (x + 5);
        printf ("in: %d  out: %d\n", x + 5, y);
    }/* end for */

    printf ("test of other pointer ...\n");
    x = 55;
    y = (*otherPtr)(x);
    printf ("x: %d y: %d\n", x, y);
    return 0;
}/* end main */
```

Notice how we declared the array:

```
int (*ptrsToFn[5])(int)
```

This declares that the variable **ptrsToFn** is an array of pointers to functions which will require a single integer parameter, and return an integer. We know that **ptrsToFn** is an array, as the square brackets after the variable name indicate that this variable is an array of 5 objects, that happen to be pointers to functions that return **int** data objects, and takes a single **int** as a parameter.

The values we initialized this array with are the addresses of five different functions. Note that we prototyped the functions before their use in the array initialization. This is

needed to satisfy the compiler that what we are initializing this array with are indeed functions. This program also initializes a simple pointer to a function variable called **otherPtr**.

To use these pointer variables and arrays, notice in the **main** function how we call the five functions in a loop:

```
y = (ptrsToFn[x]) (x + 5);
```

We use the variable **x** as an index into the array, and set the variable **y** equal to the return value of the **x**th entry in that array. The parameter we pass to each of the five functions is **x + 5**. The use of the round brackets around **ptrsToFn[x]** is not required in all compilers since most compilers can resolve the fact that this is a reference to an array of function pointers, and that an array reference will extract a value out of the array, which in this case is simply an address of code.

The last example shows our normal pointer to a function. Here we use the contents of operator to extract the address of a function from a pointer, and use this address to indirectly call the function in question. Again, note the use of round brackets to ensure our compiler knows that we are working with the contents of a function pointer. (Not all older compilers are able to cleanly compile without the brackets).

## 26.8. Other Advanced Uses

Pointers to functions are primarily used in situations like menu handlers. A menu data object is declared, usually as a structure which contains the menu text, the number of sub menu items, and a pointer to the function that handles the menu options. These function pointers are useful to declare the handlers for menu items, thus, an overall menu function need not worry about what context the menu item is being called under. It will simply use the supplied pointer and call the function indirectly.

Microsoft Windows, and other graphical operating systems, also use pointers to functions in interesting manners. Windows, in particular, requires the address of functions so that Windows can use these functions in a **call back** scheme. Most operations are done in a function called **WinProc**, which is the main Windows procedure handler for your application. But in a typical Windows program, **WinProc** is never called directly. In fact, Windows is informed of the address of a particular application's **WinProc**, and Windows will call **WinProc** itself whenever it sees fit!

## 26.9. Common Beginner Errors

- Be **very** careful with the syntax for pointers to functions.

## 26.10. Summary

This chapter showed some interesting and advanced uses of pointers. Pointers are not limited to just containing addresses of data objects, but can contains address of other pointers, as well as address of other functions in our program. The name of a function is a

pointer to the first instruction in that function in the code memory of our program, and function pointer declarations can be built up as simple variables as well as more complex arrays. In fact, a function pointer can be used in virtually all data objects including structures.

# Chapter 27. Now what?

By now, you should have a very good idea of how to write good, well-designed, clear code in the C language. Some day, you'll be able to write good, well-designed, extremely unclear code like these winners from the International Obfuscated C Code Contest (http://www0.us.ioccc.org) ...

## 2000 Winner for Most Specific Output

**Code:**

```
int m = 754974721, N, t[1 << 22], a, *p, i, e = 1 << 22, j, s, b, c, U;
f (d)
{
  for (s = 1 << 23; s; s /= 2, d = d * 1LL * d % m)
    if (s < N)
      for (p = t; p < t + N; p += s)
      for (i = s, c = 1; i; i--)
        b = *p + p[s], p[s] = (m + *p - p[s]) *
          1LL * c % m, *p++ = b % m, c = c * 1LL * d % m;
  for (j = 0; i < N - 1;)
    {
      for (s = N / 2; !((j ^= s) & s); s /= 2);
      if (++i < j)
      a = t[i], t[i] = t[j], t[j] = a;
    }
}

main ()
{
  *t = 2;
  U = N = 1;
  while (e /= 2)
    {
      N *= 2;
      U = U * 1LL * (m + 1) / 2 % m;
      f (362);
      for (p = t; p < t + N;)
      *p++ = (*p * 1LL ** p % m) * U % m;
      f (415027540);
      for (a = 0, p = t; p < t + N;)
      a += (6972593 & e ? 2 : 1) ** p, *p++ = a % 10, a /= 10;
    }
  while (!*--p);
  t[0]--;
  while (p >= t)
    printf ("%d", *p--);
}
```

**Author:** Fabrice Bellard (http://www.enst.fr/~bellard)

**Author's Comment:** This program prints the biggest known prime number (2^6972593-1) in base 10. It requires a few minutes. It uses a Modular Fast Fourier Transform to compute this number in a reasonable amount of time (the usual method would take ages!).

## 2000 Winner for Best Use Of Flags

**Code:**

```
#include <stdio.h>

char
*T="IeJKLMaYQCE]jbZRskc[SldU^V\\X\\|/_<[<:90!\"$434-./2>]s",
K[3][1000],*F,x,A,*M[2],*J,r[4],*g,N,Y,*Q,W,*k,q,D;X(){r  [r
[r[3]=M[1-(x&1)][*r=W,1],2]=*Q+2,1]=x+1+Y,*g++=((((x&     7)
-1)>>1)-1)?*r:r[x>>3],(++x<*r)&&X();}E(){A||X(x=0,g       =J
),x=7&(*T>>A*3),J[(x[F]-W-x)^A*7]=Q[x&3]^A*(*M)[2         +(
x&1)],g=J+((x[k]-W)^A*7)-A,g[1]=(*M)[*g=M[T+=A            ,1
][x&1],x&1],(A^=1)&&(E(),J+=W);}l(){E(--q&&l              ()
);}B(){*J&&B((D=*J,Q[2]<D&&D<k[1]&&(*g++=1                ),
!(D-W&&D-9&&D-10&&D-13)&&(!*r&&(*g++=0)                   ,*
r=1)||64<D&&D<91&&(*r=0,*g++=D-63)||D                     >=
97&&D<123&&(*r=0,*g++=D-95)||!(D-k[                       3]
)&&(*r=0,*g++=12)||D>k[3]&&D<=k[                          1]
-1&&(*r=0,*g++=D-47),J++));}j(                            ){
putchar(A);}b(){(j(A=(*K)[D*                              W+
r[2]*Y+x]),++x<Y)&&b();}t                                  ()
{(j((b(D=q[g],x=0),A=W)                                    ),
++q<(*(r+1)<Y?*(r+1):                                      Y)
)&&t();}R(){(A=(t(                                         q=
0),'\n'),j(),++r                                           [2
]<N)&&R();}O()                                             {(
j((r[2]=0,R(                                               ))
),r[1]-=q)                                                 &&
O(g-=-q)                                                   ;}
C(){(                                                      J=
gets                                                       (K
[1]))&&C((B(g=K[2]),*r=!(!*r&&(*g++=0)),(*r)[r]=g-K[2],g=K[2
],r[
1]&&
O())
);;}
main
(){C
((l(
(J=(
A=0)
[K],
A[M]
=(F=
(k=(
M[!A
]=(Q
=T+(
```

```
q=(Y
=(W=
32)-
(N=4
))))
+N)+
2)+7
)+7)
),Y=
N<<(
*r=!
-A))
);;}
```

**Author:** Glyn Anderson

**Author's Comment:** This program is a simple filter that translates ASCII text into semaphore code. It reads from standard input and converts all alphanumeric characters to their corresponding semaphore codes.

Consecutive whitespace is translated as a single 'break' code. Newlines also cause the next code to be output on a new line. All other characters are ignored. The codes themselves are written to standard output as a depiction of a person holding two flags, using only ASCII characters (semaphore smileys, if you will). For example, given the input "Hello, world!", the output will be:

```
            <>      <>      <>   <>                <>    <>
  _()      ()/     ()/     ()/    _\)      ()      (/_    _\)
[] /^     |^^     /^^     /^^   [] ^^     |^^|     ^^ [] [] ^^
  <>[     ][    <>][    <>][      ][     [][]     ][      ][


            <>   []
  _()_     ()/    |()      ()
[] ^^ []  /^^      ^^|    |^^|
   ][    <>][      ][]    [][]
```

## 1996 Winner for Best Output

**Code:**

```
float s=1944,x[5],y[5],z[5],r[5],j,h,a,b,d,e;int i=33,c,l,f=1;int g(){return f=
(f*6478+1)%65346;}m(){x[i]=g()-l;y[i]=(g()-l)/4;r[i]=g()>>4;}main(){char t[1948
]=" `MYmtw%FFlj%Jqig~%`jqig~Etsqnsj3stb",*p=t+3,*k="3tjlq9TX";l=s*20;while(i<s)
p[i++]='\n'+5;for(i=0;i<5;i++)z[i]=(i?z[i-1]:0)+l/3+!m();while(1){for(c=33;c<s;
c++){c+=!((c+1)%81);j=c/s-.5;h=c%81/40.0-1;p[c]=37;for(i=4;i+1;i--)if((b=(a=h*x
[i]+j*y[i]+z[i])*a-(d=1+j*j+h*h)*(-r[i]*r[i]+x[i]*x[i]+y[i]*y[i]+z[i]*z[i]))>0)
{for(e=b;e*e>b*1.01||e*e<b*.99;e-=.5*(e*e-b)/e);p[c]=k[(int)(8*e/d/r[i])];}}for
(i=4;i+1;z[i]-=s/2,i--)z[i]=z[i]<0?l*2+!m():z[i];while(i<s)putchar(t[i++]-5);}}
```

**Author: Thor AAge Eldby**

**Author's Comments:** This program shows flying spheres. The program eats CPU on lesser equipped systems. The program will run until termination. Terminal must be vt100 or better with 80 columns and 24 rows or more.

## Disclaimer!

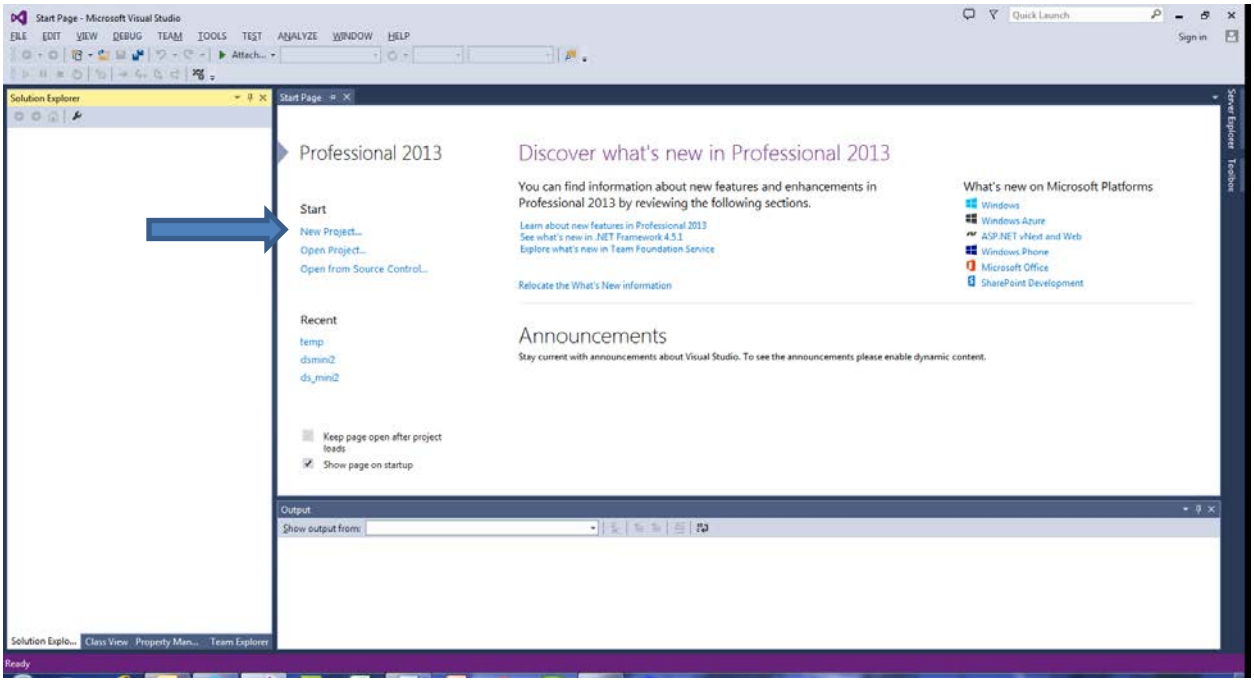
These programs (and any others that you'll find on the website) will probably need a very specific system in order to compile and run (usually gcc under UNIX).

## Another Disclaimer!

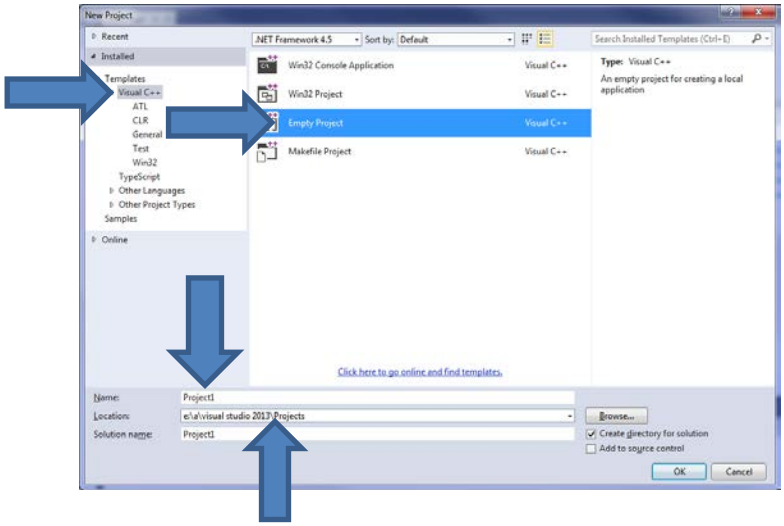
Don't even **think** about submitting code that looks like this as a code sample when you're interviewing ...

# Appendix A. Setting Up A Visual Studio Project

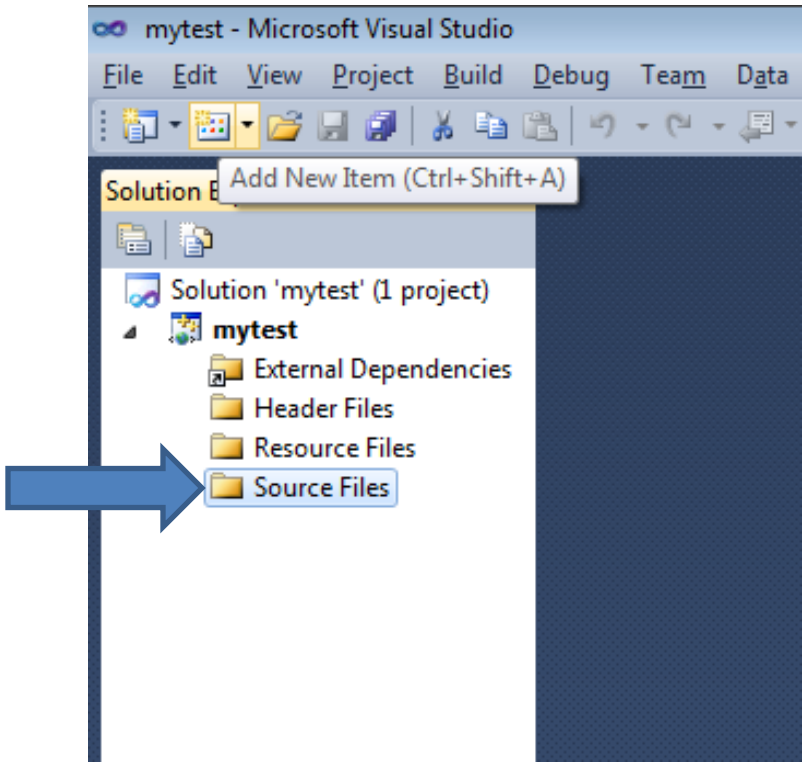
- Start up Visual Studio 2013 Professional (the instructions are basically the same for other versions, the windows just look a tiny bit different).
- Choose *New Project* (on the *Start Page*; alternatively, select FILE / NEW / PROJECT in the menus).



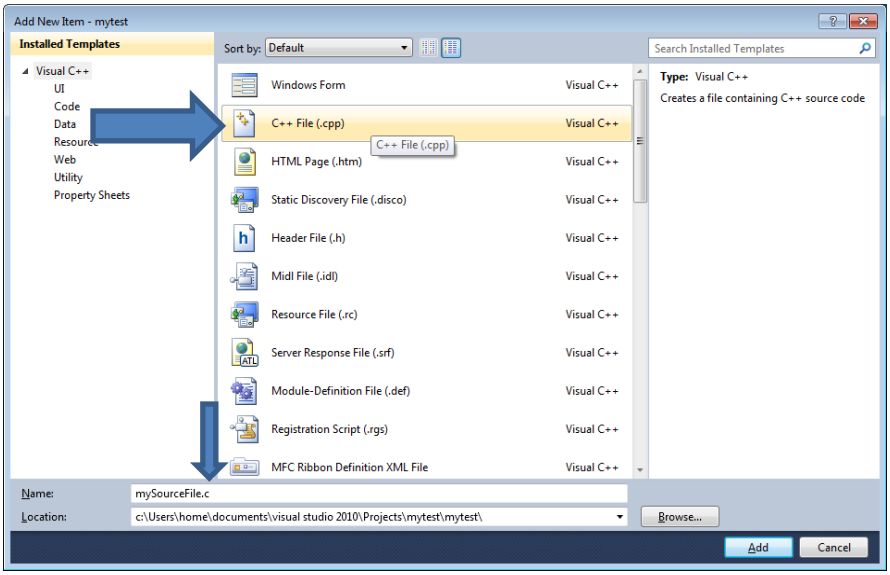

- Under *Installed / Templates* in the left-side pane, choose Visual C++. In the middle pane, click on *Empty Project*. Also enter a *Name* for the project in the textbox under the main windows. Make sure that the *Location* matches with where you want to store the project's directory. Do **not** enter a network location.



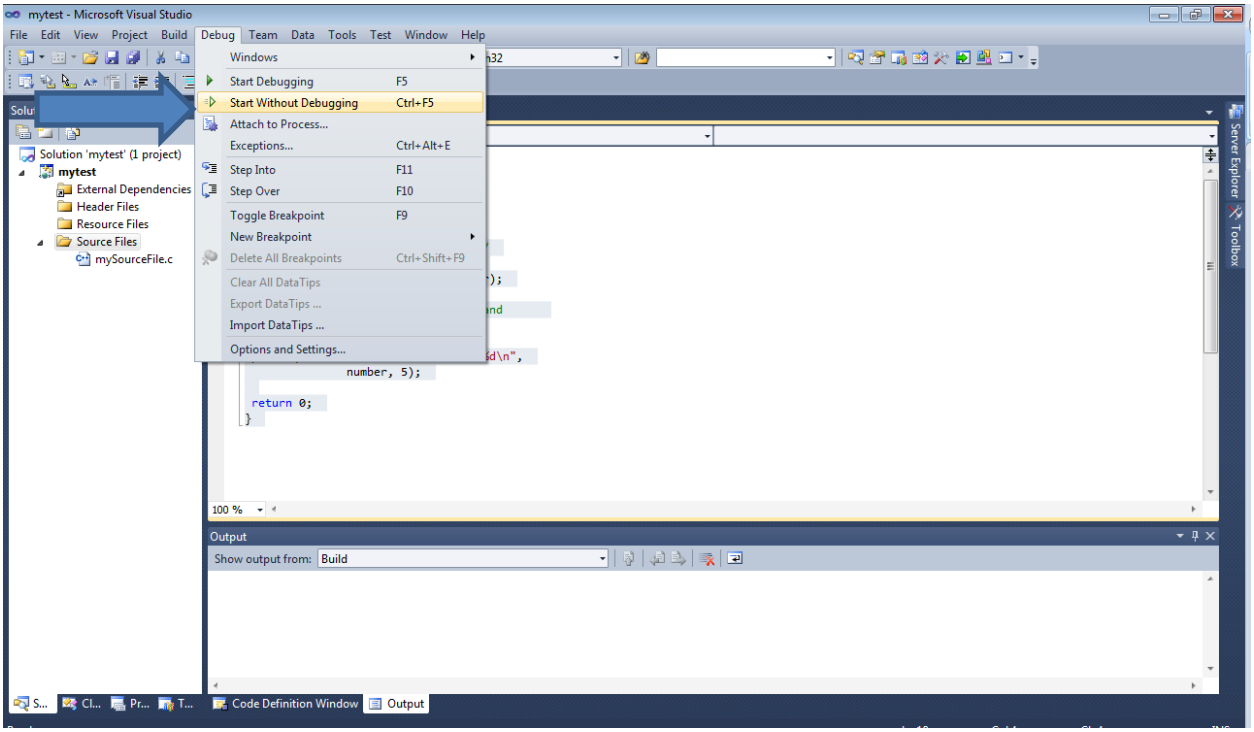
- This will bring up an empty project. We need to create a source file within it. In the *Solution Explorer* pane on the left, expand the name of the project, right-click on *Source Files*, and choose *Add*. Then choose *New Item.*



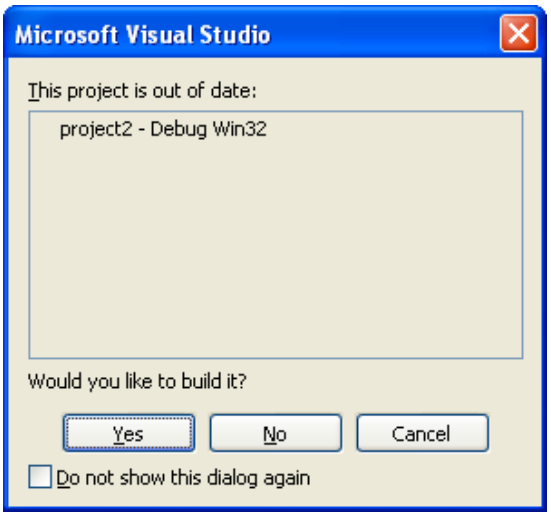

- In the right-hand pane, select *C++ File*. Also give the file a name in the *Name* text box. Make sure that the name ends in .c. Yes, I'm serious about this.



- Type your source code in the source window. Once you're ready to try it out, click on *Start Without Debugging* in the *Debug* menu.




- You might get a dialog box that looks like this:




- If you get the above dialog box, select Yes. Your code will then compile and run. It is also highly recommended that you check the "Do not show this dialog again" checkbox.

# Appendix B. Counting Systems

## Decimal

The normal method of counting for a human being is based on a numbering system which is usually referred to as **decimal**. This counting system, as we all know, is founded upon the fact that there are 10 distinct digits in the numbering system. The decimal or base 10 counting system counts in the following manner:

```
0 1 2 3 4 5 6 7 8 9
```

After 9, we put a zero in the **one's** place, and put a 1 in the **ten's** place obtaining the number 10. We proceed to count:

```
11 12 13 14 15 16 17 18 19 20
```

and so on.

Notice that each time we count out 10 distinct digits (from 0 to 9), we add one to the next placeholder to the left of the digit we have just incremented. This is how we proceed from 9 to 10, from 19 to 20, from 99 to 100 and so on. The time we add one to the placeholder to the left is when the current placeholder **overflows** its prescribed range of values, as what happens when 9 changes to a 10.

Although these concepts appear absolutely trivial to anyone that can grasp the significance of pointers in C (a bit of sarcasm, for the humour impaired!), the idea of incrementing a placeholder until it overflows its range is vitally important to proper understanding of the next two counting systems, **binary** and **hexadecimal**.

## Binary

Binary numbering systems are, as the name suggests, base 2. That is, there are only two different values to count before we overflow the prescribed range of values. The binary counting system works as follows:

```
0 1 10 11 100 101 110 111 1000 1001
```

and so on.

Notice that when we reach a value of 1 in the current placeholder, and we increment this placeholder, we will overflow the placeholder, and we replace it with zero and add one to the placeholder to the left. When we normally would have reached the value 2 in decimal,

we now get 10 in binary.

The equivalents of binary to decimal can be summarized in the following table:

| *Binary* | *Decimal* |
|---|---|
| **00000000** | 0 |
| **00001000** | 8 |
| **00001001** | 9 |
| **00010000** | 16 |
| **00100000** | 32 |
| **00100001** | 33 |
| **10000000** | 128 |
| **11111111** | 255 |

Although by no means complete, the table above should give you an idea of how the binary counting system works. Normally, a programmer who wants to work with binary will try to pad the binary number with **leading zeros** as was done above so that the binary number reflects the correct number of **bits** a data object is capable of containing. **Remember that a byte is composed of 8 bits.**

The relevance of bits to binary is the fact that a bit is capable of storing two different (or binary) values: 0 or 1, on or off, true or false, etc. The values a bit can represent are usually values that are total opposites of each other, so concepts such as high or low, good or bad and so on are other valid binary values that a bit can hold.

The different data types in C are composed of multiples of bytes, and as stated above, a byte contains 8 bits. For a byte, the total number of different values that can be represented by 8 bits are 2 to the power 8, or 256 different values, ranging from 0 to 255 (for **unsigned chars**) or -128 to 127 (for **chars**).

For an **int**, the number of values is 2 to the power 16, or over 65000 different values. A long is 2 to the power 32, or well over 4 billion! As you can see, the number of different values is rising exponentially, since we are raising 2 to some exponent based upon the number of bits in the data object. A short form method of writing 2 to the power (value) is to write 2 ^ (value).

As a side note, bits are numbered starting from bit zero which is the first bit on the right. For a byte, bit 7 is the first bit on the left, for an integer, bit 15 is the first bit on the left. Oftentimes, programmers will refer to the first bit on the left as the **high bit**.

To convert any decimal value into binary, see what powers of 2 can be added together to make the decimal value. For instance:

```
57 is equivalent to: 2 ^ 5 (32)
+ 2 ^ 4 (16)
+ 2 ^ 3 (8)
+ 2 ^ 0 (1)

giving a bit representation of:

00111001
```

A concept that lends itself quite nicely to binary are the **bitwise** operators: **!** | and ^, as well as **~**. Since these operators do things to the **bits** within a data object, understanding how bits are arranged in a data object will make the above operators obvious in their operation.

For the bitwise complement (~), we simply flip all 1 bits to 0 bits, and all 0 bits to 1 bits. Thus:

```
char x, y;
. . .
x = 63; /* 00111111 in binary */
y = ~ x;
```

will cause variable **y** to contain **11000000** in binary, which is equivalent to 192. Notice that if we add 63 and 192 we obtain 255. A bitwise complement can also be computed by simply subtracting the value in question from the largest value the data object can hold. For a byte, ~63 is simply 255 - 63, or 192. For ~65535, we get 0, and so on. Simple!

One trick programmers like to use is to define the following constants:

```
#define FALSE   0
#define TRUE    ~FALSE
```

Here, **FALSE** is given the value of zero, and **TRUE** is given the value of ~0, which is 65535, (or negative 1 if you are using signed arithmetic). In C, false is always zero, and true is anything but zero, so the definitions above are valid.

The **&** operator (when not prepended in front of a variable) is a bitwise AND. This is a binary operator, and needs two operands to work with. The bitwise AND will take the bits in both operands, and AND them together. Two 1's make a 1, otherwise we get a zero. Thus:

```
00101010
& 11100110 makes
00100010
```

The binary AND operator is a wonderful method of testing for the presence of a certain bit in a value. If we wanted to check to see if the second bit from the right is on inside a value, perform the following:

```
if (value & 2) /* 2 is 00000010 */
    statement;
```

If the value has the second bit from the right turned on, the bitwise AND will leave that bit set in the result, and because the result is non-zero, **statement** will execute!

Bitwise OR (|) is similar except that if both bits are 0's, we generate a 0, otherwise generate a 1. Bitwise ORs are excellent for setting particular bits in a value. For instance:

```
value |= 4; /* 4 is 00000100 */
```

will turn on the 3rd bit from the right in the variable **value**.

The bitwise XOR operator (^) is used to eXclusive OR two values together. XOR is the same as OR, except in the case where two bits are 1's. This case will generate a 0 using XOR when it generated a 1 with an OR.

Another closely coupled set of operators are the bitwise shifting operators >> and <<. These operators will move bits within a data object back and forth by the specified amount of bits. For example:

```
X = 1 << 4;
```

will set **x** equal to (binary) **00000001** shifted to the left by 4 bits, which will result in **00010000**, or 16. Shifting to the right is similar in nature, except the bits move to the right. For example, 33 (**00100001**) >> 2 is: **00001000** (the value 8). The shift of 1 by 4 bits is like setting **x** to 1, and multiplying by 2 four times. (1 * 2 = 2, 2 * 2 = 4, 4 * 2 = 8, 8 * 2 = 16), and a shift left of 2 is an integer divide of the value by 2 twice (33 / 2 = 16, 16 / 2 = 8). In fact, binary shifts are **very** fast multiplies and divides by the value 2. For each shift done, we are either multiplying by 2 (shift left) or integer dividing by 2 (shift right). Most modern compilers will recognize the fact that you may be doing powers of 2

for divides and multiplies, and will compile these operations as shifts for you for optimum speed from your compiled code!

## Hexadecimal

The most significant counting system of interest to programmers is the **hexadecimal** system, or **hex** for short. This system uses yet another base to count by, this time, base 16.

This is a particularly odd looking counting system since we need 16 different symbols for each placeholder. For decimal, we only needed 10, and the ten symbols are the numbers 0 through 9. For hex, we use the same 10 number symbols, and add the following symbols:

```
A B C D E F
```

When we reach F, and we add one more, we find we have exhausted the range of the first placeholder, thus we must increment the next placeholder to the left. The net result is we get, none other than, 10!

The first few counts in hex are:

```
 0  1  2  3  4  5  6  7
 8  9  A  B  C  D  E  F
10 11 12 13 14 15 16 17
18 19 1A 1B 1C 1D 1E 1F
20 21 22 23 24 25 26 27
```

and so on.

The interesting thing about hex is that we can represent all values that a byte can hold using only 2 placeholders in hex. It turns out that the value 255 is FF in hex.

In C, to differentiate between a decimal number and a hex number, we prepend all hex constants with the characters:

```
0x
```

which informs the compiler that a hex number is about to be defined. FF in hex will be written as **0xFF** in a C program. C compilers will accept either 0xFF or 0xff for the same value of 255 (decimal).

Some notable numbers in hex are:

```
0x00 - 0 0x10 - 16 0x20 - 32
0x40 - 64 0x80 - 128 0xFF - 255
```

Note that 0xFF plus 1 is 0x100, as is hopefully expected.

To convert a base 10 number into hex, take the base 10 decimal number, and divide by 16. We keep dividing the result by 16 until we get a quotient less than 16. The quotient that we obtain (the whole number portion of the division result) is the first hex digit.

Take the first hex digit, and multiply it with 16 by the number of divides we did, and subtract that result from the original number. The remaining value is taken through the above process to extract the second hex digit, and the same for the third, and so on. For example, the value 1300 in decimal converts into:

```
1300/16=81.25/16 = 5
1300-5*16*16=20/16 = 1
20 - 1*16 = 4
```

Therefore, 1300 in decimal is 0x514 in hex. Or even easier, use your Windows calculator application, and switch into scientific mode, which has binary, decimal and hexadecimal conversions available.

A question must be burning in everyone's heads by now: "why do we need to know this weird stuff?" The answer is that virtually all computers, deep down inside of their microprocessors and memory, all think naturally in base 16 and binary. Most hardware for a computer is communicated to using hex and binary values. Even a little knowledge about hex and binary will enable you to appreciate what some of the strange values your PC's hardware manuals mean!

Octal

There is one other counting system that may be of interest: the **octal** system, which is base 8. By this point, it should be apparent that base 8 is counting like the following:

```
0 1 2 3 4 5 6 7
10 11 12 13 14 15 16 17
20 21 22 . . .
```

and so on.

# Appendix C. Recap of Course Requirements Mentioned In This Document

## Header Comments

You must have a header comment in each source code file that you create. It must contain the information mentioned above. The format should be similar to that shown in the example above.

## Header Comment Description

The description contained within the header comment should accurately describe the intent of the contents of the file. This is most important in multi-file projects that contain many, many .c and .h files. Thus, a description like “This is assignment #2.” is bad while a description like “This program takes Fahrenheit temperatures from the user and displays them as Kelvin and Celsius temperatures. The program loops until the user enters an invalid temperature.” is good.

## main() Return Value

The main() function must return an int value. If you don't care what the value is (and you typically don't), you can feel free to use return 0 before the closing brace in main().

Don’t return void! Don't leave out the return value!

## Style Consistency

You must use correct and consistent style in this course!

## Distinguishing Between Words In Identifiers

Use camel case to distinguish between words in names, unless otherwise explicitly stated.

## Naming

Start your names with a lowercase letter, unless otherwise explicitly stated.

## Initialization of Variables

Always initialize your variables when you declare them, unless notified otherwise. Failing to do so will cause you to lose marks and get quite upset about it.

## Global Variables Banned

Do not use global variables unless explicitly permitted by the instructor. Note that this does not restrict the use of global constants (variables whose declarations are preceded by the keyword const; this is to be discussed later).

There is a LARGE mark penalty that is associated with using global variables when not allowed to.

## Indentation After Decisions

You must indent the body of the if statement to show that a decision has to be made and the indented code may or may not be executed. This also applies to while, do-while, and

for. The principle is to indent the body after a decision. This also implies that the decision and body cannot be on the same line.

## Braces on Single Statement Bodies

You are required to put braces on single-statement bodies in this course.

## Recursive Calls of main()

It is typically a very bad idea to recursively call main(). You will lose marks if you do so.

## Function Comments

You must have function comments for all functions you create except main(). This comment must be immediately before the function definition.

## Prototyping

You must prototype all functions that you write except main() (which has its own implicit prototype). You must also provide void in the parameter list if there are no parameters. Failing to do so will lose you marks.

## Naming Standards

Take a look at the SET Coding Standards to see what is required.

## goto

Never use a goto statement. If you do, you will lose TONS of marks.

## Always Initialize Arrays

Always initialize your array variables, even if it's just with a single 0 in the very first element.

## Pointer Initialization

Always initialize pointer variables. If you don't know what a valid value would be for the pointer when you declare it, initialize it to NULL.

## scanf()

Do not use scanf(). Use sscanf() (or one of the conversion functions like atoi(), atol(), or atof()) instead, in conjunction with gets() or fgets(). Failing to do so will lose you marks!

## Check NULL

Whenever a function can return NULL to indicate an error, check the return value.

## Freeing Memory

Whenever allocating memory, always free up memory you do not require, after you have completed your memory operations. Do not assume that the operating system will do this for you.

## Closing files

Most operating systems will automatically close opened files when the application terminates. This is not the most appropriate technique to use, for the same reasons as for dynamic memory. To ensure full portability of software across all operating systems, you must always manually close your files when no longer required, to avoid potential operating system deficiencies.