

INTRODUCTION TO PROGRAMMING



Prepared by:

Mahmoud El-Gayyar

Computer Science Department

Faculty of Computers and Informatics

Suez Canal University

2014

Table of Contents

1	Introduction to Programming	6
1.1	Programming Skills	6
1.2	The Programming Model	7
1.2.1	Problem Analysis and Specification	8
1.2.2	Algorithm Development	8
1.2.3	Program Coding	9
1.2.4	Program Execution and Testing	9
1.3	Elements of Real Programming Languages	9
1.4	Characters, Strings, and Numbers	11
1.5	History of C	12
1.6	Higher Level Languages	12
1.7	Compiler Terminology	13
1.8	Exercises	14
2	Introduction to C Language	15
2.1	Your First C Program	15
2.2	Your Second C Program	17
2.3	Program Structure	20
2.4	Exercises	23
3	Basic Data Types and Operators	25
3.1	Types	25
3.2	Constants	26
3.3	Declarations	28
3.3.1	Variable Names	29
3.4	Operators	30
3.4.1	Arithmetic Operators	30
3.4.2	Assignment Operators	32
3.5	Function Calls	33
3.6	Exercises	34
4	Statements and Control Flow	36
4.1	Expression Statement	36
4.2	if Statements	37
4.3	Switch Statement	42
4.4	Boolean Expressions	44
4.5	While Loops	47
4.6	For Loops	49
4.7	Break and Continue	52
4.8	Exercises	54
5	More about Declarations and Operators	61
5.1	Arrays	61
5.1.1	Array Initialization	64
5.1.2	Arrays of Arrays (Multidimensional Arrays)	65
5.2	More Operators	67
5.2.1	Assignment Operators	68
5.2.2	Increment and Decrement Operators	69
5.2.3	Order of Evaluation	72
5.3	Exercises	76

6	Functions and Program Structure	80
6.1	Functions Basics	80
6.1.1	Function Prototypes	84
6.1.2	Function Philosophy	85
6.2	Void (Non Value-Returning) Functions	86
6.3	Variables Visibility and Lifetime	87
6.3.1	Default Initialization	91
6.3.2	Examples	91
6.4	Exercises	93
7	Basic Input and Output	96
7.1	“printf” Function	96
7.2	Character Input and Output	99
7.2.1	Reading Lines	102
7.2.2	Reading Numbers	105
7.3	Strings	106
7.4	Exercises	112
8	User Defined Types: Structures	116
8.1	Structures	116
8.2	Accessing Members of Structures	119
8.3	Operations on Structures	120
8.3.1	Nested Structures	121
8.3.2	Arrays of Structures	122
8.4	Define a Type New Name: typedef	122
8.5	Exercises	124
9	Pointers and Memory Allocation	125
9.1	Basic Pointer Operations	126
9.1.1	Pointers and Arrays: Pointer Arithmetic	131
9.1.2	Pointer Subtraction and Comparison	133
9.1.3	Null Pointers	134
9.2	Pointers and Passing Arguments	136
9.3	Memory Allocation	138
9.3.1	Allocating Memory with malloc	138
9.3.2	Freeing Memory	142
9.3.3	Reallocating Memory Blocks	143
9.3.4	Dynamic Memory Allocation in C++	146
9.3.5	Pointer Safety	147
9.4	Exercises	148
	APPENDIX I: Compilation of a C Program	152
	References	158

List of Illustrations

Figure 1-1: Computer-based Problem Solving	8
Figure 1-2: C Compiler Terminology	14
Figure 2-1: The Structure of a C Program	21
Figure 3-1: Example of a Variable Storage	25
Figure 5-1: Multidimensional Array	66
Figure 10-1: Visual Studio New Project	154
Figure 10-2: Win32 Application Wizard	154
Figure 10-3: Win32 Application Settings	155
Figure 10-4: Add New Item Dialog Box	156

List of Tables

Table 3-1: Basic Data Types in C..... 26

Table 3-2: Character Escapes 28

Table 3-3: Arithmetic Operators30

Table 4-1: Relational Operators..... 45

Table 4-2: Boolean Operators..... 46

Table 6-1: Variables Visibility and Lifetime 90

Table 7-1: Format Specifiers 97

List of Listings

Listing 2-1: Hello World Program	16
Listing 2-2: Print few Numbers Program.....	18
Listing 4-1: Example of an if-else Statement	38
Listing 4-2: Example of a Nested If.....	39
Listing 4-3: Nested If-Else to Choose between Alternatives	41
Listing 4-4: Example of a Switch Statement	42
Listing 4-5: Example of a While Loop	48
Listing 4-6: Printing Prime Numbers between 1 and 100	52
Listing 5-1: Example of an Array Usage	64
Listing 6-1: Example of a User-defined Function	82
Listing 6-2: Example of Variables Visibility.....	92
Listing 7-1: A Program to Copy Input to Output.....	99
Listing 7-2: Function to Read One Line	102
Listing 9-1: Wrong Swap Program	136
Listing 9-2: Correct Swap Program	137
Listing 9-3: Read Lines from a User.....	145



1 Introduction to Programming

Programming a computer simply means telling it what to do. There are no other **truly fundamental** aspects of computer programming; everything else we talk about will simply be the details of a particular mechanism (usually a computer language) for telling a computer what to do. The first hard thing about programming is to learn, become comfortable with, and accept these mechanism, whether they make “sense” to you or not.

In this introduction we'll consider several things: **programming skills**, the **programming model**, **characters and strings**, **history of C**, **higher level languages** and **compiler terminology**.

1.1 Programming Skills

I'm not going to claim that programming is easy, but I am going to say that it is not hard for the reasons people usually assume it is. Programming is not a deeply theoretical subject like Chemistry or Physics; programming does not require a special talent or skill. Programming does, however, require care and craftsmanship. Some things you do need are:

1. **Attention to detail:** In programming, the details matter. Computers are incredibly stupid. You can't describe your program 3/4 of the way and then say “You know what I mean?”
2. **Stupidity:** Computers are incredibly stupid. They do exactly what you tell them to do: no more, no less. When you're programming, it helps to be able to “think” as stupidly as the computer does, so that you're in the right frame of mind for specifying everything in minute detail, and not assuming that the right thing will happen unless you tell it to.

3. **Good memory:** There are a lot of things to remember while programming: the syntax of the language, the set of prewritten functions that are available for you to call and what parameters they take, what variables and functions you've defined in your program and how you're using them, techniques you've used or seen in the past which you can apply to new problems, bugs you've had in the past which you can either try to avoid or at least recognize by their symptoms. The more of these details you can keep in your head at one time (as opposed to looking them up all the time), the more successful you'll be at programming.
4. **Ability to abstract, think on several levels:** This is probably the most important skill in programming. Computers are some of the most complex systems we've ever built, and if while programming you had to keep in mind every aspect of the functioning of the computer at all levels; it would be an impossible task to write even a simple program.

One of the most powerful techniques for managing the complexity of a system is to divide (abstract) it into little “black box” processes which perform useful tasks but which hide some details so you don't have to think about them all the time. We compartmentalize tasks all the time, without even thinking about it. If I tell you to go to the store and pick up some milk, I don't tell you to walk to the door, open the door, go outside, open the car door, get in the car, drive to the store, get out of the car, walk into the store, etc.

1.2 The Programming Model

A computer program consists of two parts: code and data. The code is the set of instructions for performing a task, and the data is the set of “memory locations” which contain the intermediate results which are used as the program performs its calculations.

Note that the code is relatively static while the data is dynamic. Once you've gotten a program working, its code won't change, but every time you run it, it will typically be working with different data, so the memory locations will take on different values.

To write a program to ask the computer to solve a real world problem, you need at least four steps:

1. Problem analysis and specification.
2. Algorithm development.
3. Program coding.
4. Program execution and testing.

1.2.1 Problem Analysis and Specification

Most problems that are to be solved with a computer usually break down to an **input** component, a **process** component and an **output** component. Because the initial description of a problem may be somewhat vague and imprecise, the first step in the problem- solving process is to review the problem carefully in order to determine its:

- Input: what information is given and which items are important in solving the problem,
- Output: what information must be produced to determine that the problem was solved,
- Process: identifies what actions must be performed on the input in order to produce the output.

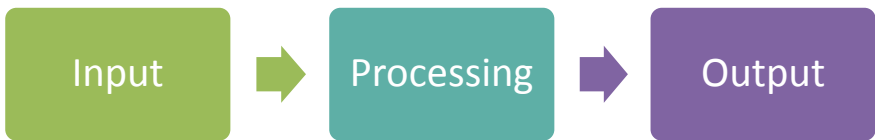


Figure 1-1: Computer-based Problem Solving

1.2.2 Algorithm Development

Once a problem has been specified, a procedure or process to produce the required output from the given input must be designed. Since the computer is a machine possessing no inherent problem-solving

capabilities, this procedure must be formulated as a detailed sequence of simple steps. Such a procedure is called an algorithm (pseudocode).

The steps that comprise an algorithm must be organized in a logical, clear manner so that the program that implements this algorithm is similarly well structured. Algorithms and programs are designed using three basic methods of control:

- **Sequential:** Steps are performed in a strictly sequential manner, each step being executed exactly once.
- **Selection:** One of several alternatives is selected and executed.
- **Repetition:** One or more steps performed repeatedly.

1.2.3 Program Coding

The third step in using the computer to solve a problem is to express the algorithm in a programming language. In the second step, the algorithm may be described in English or pseudocode, but the program that implements that algorithm must be written in the vocabulary of a programming language and must conform to the syntax of that language. The major portion of this text is concerned with the vocabulary and syntax of the programming languages C and C++.

1.2.4 Program Execution and Testing

The fourth step is to execute and test the program. This procedure should be repeated for different possible expected inputs to ensure that the program produces the expected outputs in all test cases. In case of a failure, the program can be modified, and executed again for another test run. This process continues until the program delivers the desired results.

1.3 Elements of Real Programming Languages

Programs written in a computer programming language consists of a set of several elements. If you understand these elements and what they're for, not only will you understand C better, but you'll also find learning other programming languages, and moving between different programming languages, much easier.

- 1) There are **variables**, in which you can store the pieces of data that a program is working on. Variables are the way we talk about memory locations (**data**). Variables may be **global** (that is,

accessible anywhere in a program) or **local** (that is, private to certain parts of a program).

- 2) There are **expressions**, which compute new values from old ones.
- 3) There are **assignments** which store values (of expressions, or other variables) into variables. In many languages, assignment is indicated by an equals sign; thus, we might have:

$b=3$	or	$c=d+e+1$
-------	----	-----------

Other programming languages may use other signs (like $:=$ or \leftarrow) for assignment.

- 4) There are **conditionals** which can be used to determine whether some condition is true, such as whether one number is greater than another. (In some languages, including C, conditionals are actually expressions [**logical**] which compare two values and compute a “true” or “false” value.)
- 5) Variables and expressions may have **types**, indicating the nature of the expected values. For instance, you might declare that one variable is expected to hold a number, and that another is expected to hold a piece of text. In many languages (including C), your **declarations** of the names of the variables you plan to use and what types you expect them to hold must be explicit.
- 6) There are **statements** which contain instructions describing what a program actually does. Statements may compute expressions, perform assignments, or call functions (see below).
- 7) There are **control flow** constructs which determine what order statements are performed in. A certain statement might be performed only if a condition is true. A sequence of several statements might be repeated over and over, until some condition is met; this is called a **loop**.
- 8) An entire set of statements, declarations, and control flow constructs can be lumped together into a **function** (also called routine, subroutine, method, or procedure) which another piece

of code can then call as a unit. When you call a function, you transfer control to it and wait for it to do its job, after which it **returns** to you; it may also return a value as a result of what it has done. You may also pass values to the function on which it will operate or which otherwise direct its work.

1.4 Characters, Strings, and Numbers

The earliest computers were number crunchers only, but almost all more recent computers have the ability to manipulate alphanumeric data as well. Programming languages tend to maintain a strict distinction between numbers on the one hand and alphanumeric data on the other, so we have to maintain that distinction in our own minds as well.

One fundamental component of a computer's handling of alphanumeric data is its **character set**. A character set is the set of all the characters that the computer can process and display. (Each character generally has a key on the keyboard to enter it and a bitmap on the screen which displays it.) A character set consists of letters, numbers, punctuation, etc., but the point of this discussion is not so much what the characters are but that we have to be careful to distinguish between characters, strings, and numbers.

A character is, well, a single character. If we have a variable which contains a character value, it might contain the letter 'A', or the digit '2', or the symbol '&'.

A string is a set of zero or more characters. For example, the string “and” consists of the characters 'a', 'n', and 'd'. The string “K2!” consists of the characters 'K', '2', and '!'. The string “.” consists of the single character '.', and the empty string “” consists of no characters at all.

The last two examples illustrate some important and perhaps surprising or annoying distinctions. The character '4' and the string “4” are conceptually different, and neither of them is quite the same as the number 4. The string “123” consists of three characters, and it looks like the number 123 to us, but as far as the computer is concerned it is just a string. The number 123 is, when used for ordinary numeric purposes, not represented internally as a string of three characters (instead, it is typically represented as a 16- or 32-bit integer). When we have a string which contains a numeric value which we wish to manipulate as a number, we must typically ask for the string to be explicitly converted to that number somehow. Similarly, we may have reason to convert a number to a string of digits making up its decimal representation.

1.5 History of C

Dennis Ritchie of Bell Labs created C in 1972. He and Ken Thompson worked on designing the UNIX operating system. C came from Thompson's B language. C was created as a tool for working systems programmers that needed a more readable programming language than assembler but still needed the low level access capabilities of an assembler.

C has rapidly become one of the most important and popular programming languages. Most of the UNIX operating system and MS-DOS are written in C as are most compilers and other systems and applications software.

1.6 Higher Level Languages

C is often called a middle-level computer language. Middle-level does not mean C is less powerful, harder to use, or less developed than high level languages such as BASIC or Pascal; nor is C similar to a low-level language such as assembly language. C combines elements of a high-level language with the functionalism of an assembler.

- High Level
 - BASIC, FORTRAN, Pascal, Java
- Middle Level
 - C, C++
- Low Level
 - Assembler

C and C++ were first used for systems programming. Systems programming refers to a class of programs that either are part of or work closely with the operating system of the computer.

C and C++ are used for systems programming when:

- The program must run quickly; C and C++ programs run almost as fast as ones in assembler.

- C and C++ is a programmers language, it lacks restrictions and easily manipulates bits, bytes, and memory addresses.
- A programmer needs direct control of I/O and memory management functions that C and C++ gives.

1.7 Compiler Terminology

C is a compiled language. This means that the programs you write are translated, by a special program called a **compiler**, into executable machine-language programs which you can actually run. Executable machine-language programs are self-contained that means that you don't need copies of the source code (the original programming-language text you composed) or the compiler in order to run them; you can distribute copies of just the executable and that's all someone else needs to run it.

The main alternative to a compiled computer language is an **interpreted** one, such as BASIC. An interpreted language is interpreted (by a program called an **interpreter**) line by line and its actions performed immediately. If you gave a copy of an interpreted program to someone else, they would also need a copy of the interpreter to run it. No standalone executable machine-language binary program is produced.

In other words, for each statement that you write, a compiler translates into a sequence of machine language instructions which does the same thing, while an interpreter simply perform.

When you're working with a compiled language, there are several mechanical details which you'll want to be aware of. You create one or more **source files** which are simple text files containing your program, written in whatever language you're using. You typically use a text editor to work with source files. You supply each source file (you may have one or more than one) to the compiler, which creates an **object file** containing machine-language instructions corresponding to your program. Your program is not ready to run yet, however: if you called any functions which you didn't write (such as the standard library functions provided as part of a programming language environment), you must arrange for them to be inserted into your program, too. The task of combining object files together, while also locating and inserting any library functions, is the job of the **linker**. The linker puts together the object files you give it, noticing if you call any functions which you haven't supplied and which must therefore be library functions. It then searches one or more library files (a library file is simply a collection of

object files) looking for definitions of the still-unresolved functions, and pulls in any that it finds. When it's done, it either builds the final, **executable file**, or, if there were any errors (such as a function called but not defined anywhere) complains.

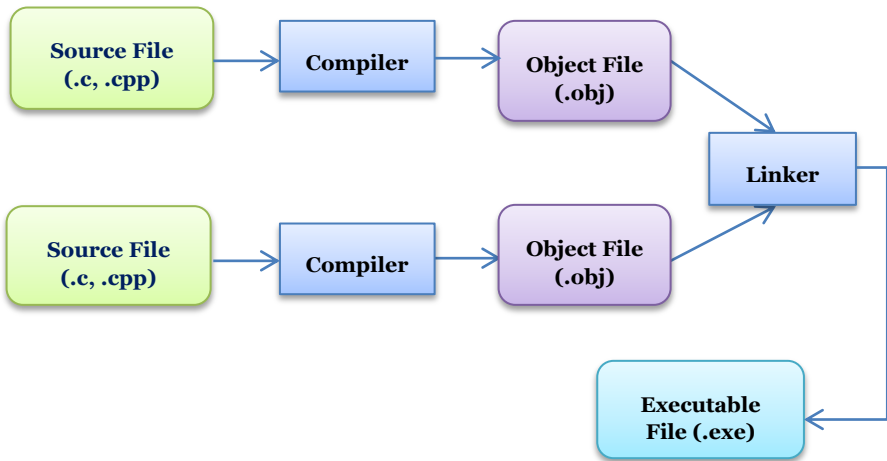


Figure 1-2: C Compiler Terminology

1.8 Exercises

1. What are the basic skills a programmer should have?
2. Explain the program life cycle?
3. What are the different types of programming languages?
4. What is the difference between the compiler and the interpreter?
5. Illustrate using a diagram the compiler terminology?



2 Introduction to C Language

A programming language is a tool, and no tool can perform every task unaided. So you have to put in your mind that C does not have built-in features to perform every function that we might ever need to do while programming. Some common tasks, such as manipulating strings, allocating memory, and doing input/output (I/O), are performed by calling on **library functions** (built-in). Other tasks which you might want to do, such as creating or listing directories, or interacting with a mouse, or displaying windows or other user-interface elements, or doing color graphics, are not defined by the C language at all. You can do these things from a C program, of course, but you will be calling on services which are peculiar to your programming environment (compiler, processor, and operating system) and which are not defined by the C standard.

Another aspect of C that's worth mentioning here is that it is a bit dangerous. C does not, in general, try hard to protect a programmer from mistakes. If you write a piece of code which will do something wildly different from what you intended it to do, up to and including deleting your data or trashing your disk, and if it is possible for the compiler to compile it, it generally will. You won't get warnings of the form "Do you really mean to...?" or "Are you sure you really want to...?". C is often compared to a sharp knife: it can do a surgically precise job on some exacting task you have in mind, but it can also do a surgically precise job of cutting off your finger. It's up to you to use it carefully. On the other hand, this point can be considered as one of the advantages of C, as you can use it to write low-level program or even computer viruses.

2.1 Your First C Program

The best way to learn programming is to dive right in and start writing real programs. This way, concepts which would otherwise seem abstract make sense, and the positive feedback you get from getting even a small program to work gives you a great incentive to improve it or write the next one.

You can't learn everything you'd need to write a complete program all at once, so you'll have to take some things “on faith” and parrot them in your first programs before you begin to understand them.

The first example program here is the first example program you will meet while learning any computer language: print or display a simple string, and exit. Here is our version of “hello, world” program:

Listing 2-1: Hello World Program

```
1. #include <stdio.h>
2. int main()
3. {
4.     printf("Hello, world!\n");
5.     return 0;
6. }
```

If you have a C compiler, the first thing to do is figure out how to type this program in and compile it and run it and see where its output went. In this course we are going to make use of Visual Studio 2010. To learn how you can use it to compile and run your programs please refer to Appendix I.

The first line is will appear in almost all programs we write. It asks that some definitions having to do with the “Standard I/O Library” be included in our program; these definitions are needed if we are to call the library function **printf** correctly.

The second line says that we are defining a function named **main**. Most of the time, we can name our functions anything we want, but the function name **main** is special: it is the function that will be “called” first when our program starts running (it the program’s **main entry point**). The empty pair of parentheses indicates that our **main** function accepts no arguments, that is, there isn't any information which needs to be passed in when the function is called. A C and C++ program must have one and only one function with the name of **main** that must be in lower case. The **main** function should have a return type of either **int** or **void**, with **int** being the preferred return type.

The braces { and } surround a list of statements in C. Here in line 3 and 6, they surround the list of statements making up the function main.

The fourth line:

```
printf("Hello, world!\n");
```

is the first **statement** in the program. It asks that the function **printf** be called; **printf** is a library function which prints formatted output on the computer screen. The parentheses surround **printf's** argument list: the information which is handed to it which it should act on. The **semicolon** (;) at the end of the line terminates the statement.

printf's first argument is the string which it should print. The string, enclosed in double quotes "", consists of the words "Hello, world!" followed by a **special sequence**: \n. In strings, any two-character sequence beginning with the backslash \ represents a single special character. The sequence \n represents the "new line" character, which ends one line of output and move down to the next.

The second statement in the **main** function is

```
return 0;
```

In general, a function may return a value to its caller, and **main** is no exception. When **main** returns (that is, reaches its end and stops functioning), the program is at its end, and the return value from **main** tells the operating system whether it succeeded or not. By convention, a return value of 0 indicates success and -1 indicates a failure.

2.2 Your Second C Program

Our second example shown in Listing 2-2 is of little more practical use than the first, but it introduces a few more programming language elements.

Listing 2-2: Print few Numbers Program

```
1. #include <stdio.h>
2. /* print a few numbers, to illustrate a simple loop */
3. int main()
4. {
5.     int i; // another way of comment
6.     for(i = 0; i < 10; i = i + 1)
7.         printf("i is %d\n", i);
8.     return 0;
9. }
```

As before, the line `#include <stdio.h>` is necessary since we're calling the `printf` function, and `main()` and the pair of braces `{}` indicate and delineate the function named `main` we're (again) writing.

The first new line is the second line:

```
/* print a few numbers, to illustrate a simple loop */
```

which is a **comment (block comment)**. Anything between the characters `/*` and `*/` is ignored by the compiler, but may be useful to a person trying to read and understand the program. You can add comments anywhere you want to in the program, to document what the program is, what it does, who wrote it, how it works, what the various functions are for and how they work, what the various variables are for, etc. Another way of comment (inline comment) is shown in line 5 that starts after the symbol `//` and ends with the end of the line.

The second new line is the fifth one:

```
int i;
```

which declares that our function will use a variable named `i`. a variable is a location in the computer's memory that may hold any value. In this case the variable type is `int`, which is a plain integer.

Next, we set up a **loop**:

```
for(i = 0; i < 10; i = i + 1)
```

The keyword `for` indicates that we are setting up a “**for loop**”. A `for` loop is controlled by three expressions, enclosed in parentheses and separated by **semicolons (;)**. These expressions say that, in this case, the loop starts by setting `i` to 0, that it continues as long as `i` is less than 10, and that after each iteration of the loop, `i` should be incremented by 1 (that is, have 1 added to its value).

Finally, we have a call to the `printf` function, as before, but with several differences. First, the call to `printf` is ***within the body of the for loop***. This means that control flow does not pass once through the `printf` call, but instead that the call is performed as many times as are dictated by the `for` loop. In this case, `printf` will be called several times: once when `i` is 0, once when `i` is 1, once when `i` is 2, and so on until `i` is 9, for a total of 10 times.

A second difference in the `printf` call is that the string to be printed, “`i is %d`”, contains a percent sign (%). Whenever `printf` sees a percent sign, it indicates that `printf` is not supposed to print the exact text of the string, but is instead supposed to read another one of its arguments to decide what to print. The letter after the percent sign tells it what type of argument to expect and how to print it. In this case, the letter `d` indicates that `printf` is to expect an `int` (integer), and to print it in decimal. Finally, we see that `printf` is in fact being called with another argument, for a total of two, separated by commas (,). The second argument is the variable `i`, which is in fact an `int`, as required by `%d`. The effect of all of this is that each time it is called, `printf` will print a line containing the current value of the variable `i`:

```
i is 0
i is 1
i is 2
...
```

After several trips through the loop, `i` will eventually equal 9. After that trip through the loop, the third control expression `i = i + 1` will increment its value to 10. The condition `i < 10` is no longer true, so no more trips through the loop are taken. Instead, control flow jumps down

to the statement following the **for** loop, which is the return statement. The main function returns, and the program is finished.

2.3 Program Structure

As observed from the last two examples, a program consists of one or more functions (Our two example programs so far have contained one function apiece). At the top of a source file are typically a few lines such as `#include <stdio.h>`, followed by the definitions (i.e. code) for the functions. (It's also possible to split up the several functions making up a larger program into several source files, as we'll see later)

Each function is further composed of declarations and statements, in that order. When a sequence of statements should act as one (for example, when they should all serve together as the body of a loop) they can be enclosed in braces (just as for the outer body of the entire function). The simplest kind of statement is an expression statement, which is an expression (presumably performing some useful operation) followed by a semicolon. Expressions are further composed of operators, variables, and constants.

C source code consists of several lexical elements. Some are words, such as **for**, **return**, **main**, and **i**, which are either keywords of the language (**for**, **return**) or identifiers (names) we've chosen for our own functions and variables (**main**, **i**). There are constants such as **1** and **10** which introduce new values into the program. There are operators such as **=**, **+**, and **>**, which manipulate variables and values. There are other punctuation characters (often called delimiters), such as parentheses and squiggly braces **{}**, which indicate how the other elements of the program are grouped. Finally, all of the preceding elements can be separated by whitespace: spaces, tabs, and the “carriage returns” between lines.

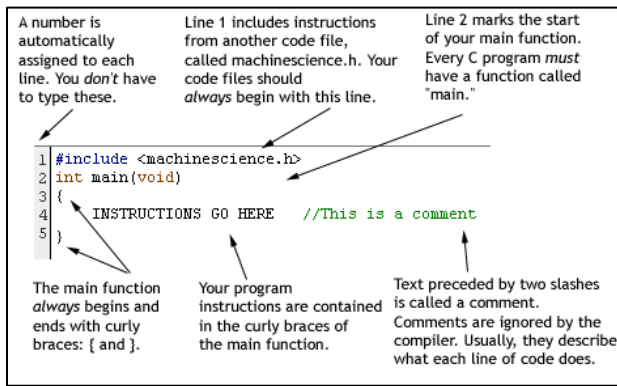


Figure 2-1: The Structure of a C Program

The source code for a C program is, for the most part, “free form”. This means that the compiler does not care how the code is arranged: how it is broken into lines, how the lines are indented, or whether whitespace is used between things like variable names and other punctuation (Lines like `#include <stdio.h>` are an exception; they must appear alone on their own lines, generally unbroken. Only lines beginning with `#` are affected by this rule). You can use whitespace, indentation, and appropriate line breaks to make your programs more readable for yourself and other people (**highly recommended**). You can place explanatory comments anywhere in your program to make it much easier to read.

To drive home the point that the compiler doesn't care about indentation, line breaks, or other whitespace, here are a few (extreme) examples: The fragments:

```
for(i = 0; i < 10; i = i + 1)

    printf("%d\n", i);
```

and

```
for(i = 0; i < 10; i = i + 1) printf("%d\n", i);
```

and

```
for(i=0;i<10;i=i+1) printf("%d\n",i);
```

and

```
    for(i = 0; i < 10; i = i + 1)
printf("%d\n", i);
```

and

```
for      (      i
= 0      ;
i <      10
; i      =
i +      1
) printf (
"%d\n"   ,      i
) ;
```

and

```
for
(i=0;
i<10;i=
i+1)printf
("%d\n", i);
```

are all treated exactly the same way by the compiler. Although C compilers do not care about how a program looks, proper indentation and spacing are critical in making programs easy for people to read.

2.4 Exercises

1. The C and C++ languages is generally considered to be a
 - a) high-level language
 - b) mid-level language
 - c) low-level language
 - d) multiple-level language
2. Which function is the entry point of a C or C++ program?
 - a) `#include "stdio.o"`
 - b) `printf()`
 - c) `#include "stdio.h"`
 - d) `main()`
3. `'\n'` represents:
 - a) a comment
 - b) a rubout character
 - c) a function
 - d) an escape sequence
4. `stdio.h` stands for:
 - a) standard input/output data handler
 - b) standard task definition input/output
 - c) standard input/output header file
 - d) standard task definition in/out handler
5. Which of the following terms is related to a location in the computer's memory that may assume any value?
 - a) constant
 - b) variable
 - c) data type
 - d) escape sequence
6. Approximately what is the line `#include <stdio.h>` at the top of a C source file for?
7. What are some uses for comments?
8. Why is indentation important? How carefully does the compiler pay attention to it?
9. What are the largest and smallest values that can be reliably stored in a variable of type `int`?
10. What is the difference between the constants `7`, `'7'`, and `"7"`?
11. What is the difference between the constants `123` and `"123"`?

12. What is the function of the semicolon in a C statement?
13. Using Appendix I, get the "Hello, world!" program to work on your computer.
14. What do these loops print?

```
for(i = 0; i < 10; i = i + 2)

    printf("%d\n", i);

for(i = 100; i >= 0; i = i - 7)

    printf("%d\n", i);

for(i = 1; i <= 10; i = i + 1)

    printf("%d\n", i);

for(i = 2; i < 100; i = i * 2)

    printf("%d\n", i);
```

15. Write a program to print the numbers from 1 to 10 and their squares:

1	1
2	4
3	9
...	
10	100



3 Basic Data Types and Operators

Before starting to program in C and C++, it is necessary to become familiar with the rules concerning the declaration of variables, the types of variables, the way those variables can be used to form expressions and the way the values for those variables can be input and output.

In general, a variable is a location in the computer's memory (RAM) that may hold a value according to its declared type. This concept is explained in Figure 3-1.

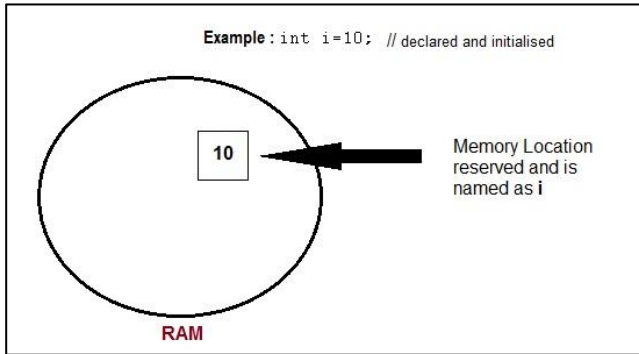


Figure 3-1: Example of a Variable Storage

3.1 Types

There are only a few basic data types in C. The first ones we'll be encountering and using are shown in Table 3-1.

The ranges listed in Table 3-1 for types `int` and `long int` are the **guaranteed** minimum ranges. On some systems (32 or 64 bits machines), these types may be able to hold larger values, but a program that depends on extended ranges will not be as portable.

Table 3-1: Basic Data Types in C

Type	No. Of bits	Minmium Value Range
char	8	0-256
int	16/32	-32,767 to 32,767
long int	32/64	-2,147,483,647 to +2,147,483,647
float	32	a floating-point number
double	64	Double a floating-point number with more precision

You might wonder how the computer stores characters. The answer involves a ***character set***, which is simply a mapping between some set of characters and some set of small numeric codes. Most machines today use the ***ASCII character set***, in which the letter A is represented by the code 65, the ampersand & is represented by the code 38, the digit 1 is represented by the code 49, the space character is represented by the code 32, etc. (Most of the time, of course, you have no need to know or even worry about these particular code values; they're automatically translated into the right shapes on the screen or printer when characters are printed out, and they're automatically generated when you type characters on the keyboard.) Character codes are usually small--the largest code value in ASCII is 126. Characters usually fit in a byte(8 bits).

Most of the simple variables in most programs are of types **int**, **long int**, or **double**. Typically, we'll use **int** and **double** for most purposes and **long int** any time we need to hold integer values greater than 32,767.

3.2 Constants

A constant is just an immediate, absolute value found in an expression. The simplest constants are decimal integers, e.g. 0, 1, 2, 123. A constant can be forced to be of type **long int** by suffixing it with the letter L (in

upper or lower case, although upper case is strongly recommended, because a lower case l looks too much like the digit 1).

A constant that contains a **decimal point** or the **letter e** (or both) is a floating-point constant: 3.14, 10., .01, 123e4, 123.456e7 . The e indicates multiplication by a power of 10; 123.456e7 is 123.456 times 10 to the power 7, or 1,234,560,000. (Floating-point constants are of type double by default.)

We also have constants for specifying characters and strings. A character constant is simply a single character between single quotes: 'A', '.', '%'. A string is represented in C as a sequence or array of characters. A string constant is a sequence of zero or more characters enclosed in double quotes: "apple", "hello, world", "this is a test".

Within character and string constants, the **backslash character** \ is special, and is used to represent characters not easily typed on the keyboard or for various reasons not easily typed in constants. The most common of these “**character escapes**” are shown in Table 3-2.

For example, “**he said \”hi\”**” is a string constant which contains two double quotes, and \” is a character constant consisting of a (single) single quote. Notice once again that the character constant 'A' is very different from the string constant "A".

C and C++ allow for the programmer to define constants that represent decimal, string and character constants. The **#define** preprocessor directive can be used to define constants that are to be used within a program.

```
#define PI 3.14156

#define MYNAME "JOHN DOE"

#define LIMIT 10
```

Table 3-2: Character Escapes

<code>\n</code>	a “newline” character
<code>\b</code>	a backspace
<code>\r</code>	a carriage return (without a line feed)
<code>\'</code>	a single quote (e.g. in a character constant)
<code>\t</code>	tab
<code>\"</code>	a double quote (e.g. in a string constant)
<code>\\</code>	a single backslash

These constants can be very useful in case that you are using a certain value several times (say 10 times) in the same program. In case that you would like to change this value, you will need to change only its definition (one line of code) rather than going through the program and changing every occurrence of this value (in this case 10 lines of code).

3.3 Declarations

A **variable** is a place you can store a value. To be able to refer to it unambiguously, a variable needs a name. You can think of the variables in your program as a set of boxes, each with a label giving its name; you might imagine that storing a value in a variable consists of writing the value on a paper and placing it in the box.

A **declaration** tells the compiler the name and type of a variable you'll be using in your program. In its simplest form, a declaration consists of the type, the name of the variable, and a terminating semicolon:

```
char c;  
  
int i;  
  
float f;
```

You can also declare several variables of the same type in one declaration, separating them with commas:

```
int i1, i2;
```

A declaration for a variable can also **contain an initial value**. This initializer consists of an equal sign and an expression, which is usually a single constant:

```
int i = 1;
```

```
int i1 = 10, i2 = 20;
```

3.3.1 Variable Names

Within limits, you can give your variables and functions any names you want. These names (the formal term is “**identifiers**”) consist of *letters*, *numbers*, and *underscores*. Theoretically, names can be as long as you want, but extremely long ones get tedious to type after a while, and the compiler is not required to keep track of extremely long ones perfectly. (What this means is that if you were to name a variable, say, supercalafragalisticespialidocious, the compiler might get lazy and pretend that you'd named it supercalafragalisticespialidocio, such that if you later misspelled it supercalafragalisticespialidociouz, the compiler wouldn't catch your mistake. Nor would the compiler necessarily be able to tell the difference if for some perverse reason you deliberately declared a second variable named supercalafragalisticespialidociouz.)

The rules for forming a variable name also apply to function names. The rules are:

- The first character must be a letter, either lowercase or uppercase;
- The capitalization of names in C is significant (case-sensitive): the variable names `variable`, `Variable`, and `VARIABLE` are all distinct;
- defined constants are **traditionally** made up of all uppercase characters
- the variable must be unique in the first eight characters in order to be safe across compilers;
- make variable names descriptive;



- do not make a variable name the same as a reserved word (the words such as **int** and **for** which are part of the syntax of the language).

3.4 Operators

Both C and C++ have a great many operators. In fact one of the criticisms of the C language is that it has too many operators which make the language difficult to read. The operators fall into several categories: arithmetic, logical, relational, bitwise, assignment and miscellaneous. In this section, we focus on arithmetic and assignment operators.

3.4.1 Arithmetic Operators

The basic operators for performing arithmetic are the same in many computer languages:

Table 3-3: Arithmetic Operators

Arithmetic Operator	Operands	Function
-	Unary	Sign (negate a number)
%	Binary	Modulus (remainder)
/	Binary	Division
*	Binary	Multiplication
-	Binary	Subtraction
+	Binary	Addition

The - operator can be used in two ways: to subtract two numbers (**Binary**) (as in $a - b$), or to negate one number (**Unary**) (as in $-a + b$ or $a + -b$).



When applied to **integers**, the division operator / discards any remainder:

$$1 / 2 \quad \rightarrow \quad 0$$

$$5 / 2 \quad \rightarrow \quad 2$$

But when either operand is a floating-point quantity (type float or double), the division operator yields a floating-point result: $1 / 2.0$ is 0.5, and $7.0 / 4.0$ is 1.75.

$$1 / 2.0 \quad \rightarrow \quad 0.5$$

$$5.0 / 2.0 \quad \rightarrow \quad 2.5$$

The modulus operator % gives you the remainder when two **integers** are divided (The modulus operator can only be applied to integers):

$$1 \% 2 \quad \rightarrow \quad 1$$

$$5 \% 2 \quad \rightarrow \quad 1$$

An additional arithmetic operation you might be wondering about is exponentiation. Some languages have an exponentiation operator (typically ^ or **), but C **doesn't**. (To square or cube a number, just multiply it by itself)

For binary operators, multiplication, division, and modulus all have higher precedence than addition and subtraction. The term “precedence” refers to how “tightly” operators bind to their operands (that is, to the things they operate on). In mathematics, multiplication has higher precedence than addition:

$$1 + 2 * 3 \quad \rightarrow \quad 7, \text{ not } 9$$

In other words,

$$1 + 2 * 3 \quad \text{is equivalent to} \quad 1 + (2 * 3)$$

C is the same way.

All of these operators “group” from left to right, which means that when two or more of them have the same precedence and participate next to each other in an expression, the evaluation conceptually proceeds from left to right.

$$1 - 2 - 3 \quad \rightarrow \quad (1 - 2) - 3 \quad \rightarrow \quad -4, \text{ not } +2$$

Whenever the default precedence doesn't give you the grouping you want, you can always use ***explicit parentheses***. For example, if you wanted to add 1 to 2 and then multiply the result by 3, you could write:

```
(1 + 2) * 3
```

3.4.2 Assignment Operators

The assignment operator = assigns a value to a variable. For example,

```
x = 1
```

sets x to 1, and

```
a = b
```

sets a to whatever b's value is. The expression

```
i = i + 1
```

is the standard programming idiom for increasing a variable's value by 1: this expression takes i's old value, adds 1 to it, and stores it back into i. (C provides several “shortcut” operators for modifying variables in similar ways, which we'll explain later.)

The assignment operator groups from right to left. Therefore,

```
c = a = b
```

is equivalent to

```
c = (a = b)
```

and assigns b's value to both a and c. It's usually a matter of style whether you initialize a variable with an initializer in its declaration or with an assignment expression near where you first use it. That is, there's no particular difference between

```
int a = 10;
```

and

```
int a;
```

```
/* later... */
```

```
a = 10;
```

3.5 Function Calls

We'll have much more to say about functions in a later chapter, but for now let's just look at how they're called. A function is a piece of code, written by you or by someone else, which performs some useful task. You call a function by mentioning its name followed by a pair of parentheses. If the function takes any arguments, you place the arguments between the parentheses, separated by commas. These are all function calls:

```
printf("Hello, world!\n")
```

```
printf("%d\n", i)
```

```
sqrt(144)
```



```
getchar()
```

The arguments to a function can be arbitrary expressions. Therefore, you don't have to say things like:

```
int sum = a + b + c;
```

```
printf("sum = %d\n", sum);
```

if you don't want to; you can instead collapse it to

```
printf("sum = %d\n", a + b + c);
```

Many functions return values, and when they do, you can embed calls to these functions within larger expressions:

```
c = sqrt(a * a + b * b)
```

```
x = r * cos(theta)
```

The first expression squares **a** and **b**, computes the square root of the sum of the squares, and assigns the result to **c**. (In other words, it computes **a * a + b * b**, passes that number to the **sqrt** function, and assigns **sqrt**'s return value to **c**.)

The second expression passes the value of the variable **theta** to the **cos** (cosine) function, multiplies the result by **r**, and assigns the result to **x**.

3.6 Exercises

1. The values placed within the parentheses of a function are called:
 - a) arguments
 - b) statements
 - c) escape sequence
 - d) include statement
2. Which of the following terms describes data that remains the same throughout a program?
 - a) constant
 - b) variable
 - c) integer
 - d) float
3. Which data type uses the most memory and provides the more precision?
 - a) float
 - b) char
 - c) int
 - d) double
4. Which data type does not support fractional values?
 - a) float
 - b) int
 - c) long float
 - d) double
5. Which data type requires only one byte of memory?
 - a) char
 - b) int
 - c) float
 - d) double

6. Which of the following are valid variable names?
- a) sam
 - b) SAM
 - c) hi_cost
 - d) 9%d4
 - e) howdoyoudothis
7. List 5 rules for forming variables names?
8. What are the two different kinds of division that the / operator can do? Under what circumstances does it perform each?
9. Evaluate the following expressions if it is possible:
- a) $10 \% 100$
 - b) $6.0 / 2$
 - c) $5 + 3 * 2$
 - d) $7.0 + (1/2)$
10. Which of the following is the correct order of evaluation for the below expression?
- $$z = x + y * z / 4 \% 2 - 1$$
- a) / % + - =
 - b) = * / % + -
 - c) / * % - + =
 - d) * % / - + =



4 Statements and Control Flow

Statements are the “steps” of a program. Most statements compute and assign values or call functions, but we will eventually meet several other kinds of statements as well. By default, statements are executed in sequence, one after another. We can, however, modify that sequence by using **control flow** constructs which arrange that a statement or group of statements is executed only if some condition is true or false, or executed over and over again to form a **loop**.

The definitions of the terms **statement** and **control flow** are somewhat circular. A statement is an element within a program which you can apply control flow to; control flow is how you specify the order in which the statements in your program are executed.

4.1 Expression Statement

Most of the statements in a C program are expression statements. An expression statement is simply an expression followed by a semicolon. The lines

```
i = 0;
```

```
i = i + 1;
```

and

```
printf("Hello, world!\n");
```

are all expression statements. In C, the semicolon is a statement terminator; all simple statements are followed by semicolons. The semicolon is also used for a few other things in C; we’ve already seen that it terminates declarations, too.

Expression statements do all of the real work in a C program. Whenever you need to compute new values for variables, you’ll typically use

expression statements. Whenever you want your program to do something visible, in the real world, you'll typically call a function (as part of an expression statement). We've already seen the most basic example: calling the function `printf` to print text to the screen. But anything else you might do--read or write a disk file, talk to a modem or printer, draw pictures on the screen--will also involve function calls.

To be useful, an expression statement must do something; it must have some lasting effect on the state of the program. (Formally, a useful statement must have at least one side effect.) The first two sample expression statements in this section (above) assign new values to the variable `i`, and the third one calls `printf` to print something out, and these are good examples of statements that do something useful. To make the distinction clear, we may note that degenerate constructions such as

```
0;
```

```
i;
```

or

```
i + 1;
```

are syntactically valid statements--they consist of an expression followed by a semicolon--but in each case, they compute a value without doing anything with it, so the computed value is discarded, and the statement is useless.

4.2 if Statements

The simplest way to modify the control flow of a program is with an *if statement*, which in its simplest form looks like this:

```
if(x > max)
```

```
    max = x;
```

Even if you didn't know any C, it would probably be pretty obvious that what happens here is that if `x` is greater than `max`, `x` gets assigned to `max`.

More generally, we can say that the syntax of an **if** statement is:

```
if( expression )  
  
    statement;
```

What if you have a series of statements, all of which should be executed together or not at all depending on whether some condition is true? The answer is that you enclose them in **braces**:

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



As a general rule, anywhere the syntax of C calls for a statement, you may write a series of statements enclosed by braces. (You do not need to, and should not, put a semicolon after the closing brace, because the series of statements enclosed by braces is not itself a simple expression statement)

An **if** statement may also optionally contain a second statement, the “else clause” which is to be executed if the condition is not met (false). An example is shown in Listing 4-1.

Listing 4-1: Example of an if-else Statement

```
1. if(n > 0)  
2.     average = sum / n;  
3. else{  
4.     printf("can't compute average\n");  
5.     average = 0;  
6. }
```

Here in this example, statement 2 is executed **if** the condition is *true*, and the block of statements 4 and 5 (following the keyword **else**) is executed if the condition is *false*. In this example, we can compute a

meaningful average only if `n` is greater than 0; otherwise, we print a message saying that we cannot compute the average. The general syntax of an `if` statement is therefore:

```
if( expression )  
    statement;    or    block  
  
else  
    statement;    or    block
```

(where block is a set statements enclosed in braces).

It's also possible to nest one `if` statement inside another. (For that matter, it's in general possible to nest any kind of statement or control flow construct within another.) For example, Listing 4-2 provides a little piece of code that contains an `if-else` statement in which the statement following the `else` is itself an `if-else` statement. If `x` is less than zero then `sign` is set to -1, however if it is not less than zero the statement following the `else` is executed. In that case if `x` is equal to zero then `sign` is set to zero and otherwise it is set to 1.

Listing 4-2: Example of a Nested If

```
1. if (x < 0)  
2.     sign = -1;  
3. else{  
4.     if (x == 0)  
5.         sign = 0;  
6.     else  
7.         sign = 1;  
8. }
```

When you have one `if` statement (or loop) nested inside another, it's a very good idea to use explicit braces `{}`, as shown, to make it clear (both to you and to the compiler) how they're nested and which `else` goes with which `if`. It's also a good idea to indent the various levels, also as shown, to make the code more readable to humans.

Novice programmers often use a sequence of **if** statements rather than use a nested **if-else** statement. That is they write the above in the logically equivalent form:

```
if (x < 0)
    sign = -1;
if (x == 0)
    sign = 0;
if (x > 0)
    sign = 1;
```



This version is not recommended since it does not make it clear that only one of the assignment statements will be executed for a given value of *x*. Also it is inefficient since all three conditions are always tested.

If nesting is carried out to too deep a level and indenting is not consistent then deeply nested **if** or **if-else** statements can be confusing to read and interpret. It is important to note that an **else** always belongs to the closest **if** without an **else**.

Here is an example of another common arrangement of **if-else** statements to choose between several alternatives. Suppose we have a variable *grade* containing a student's numeric grade, and we want to print out the corresponding letter grade. In Listing 4-3 you can find the code that would do the job.

What happens here is that exactly one of the five **printf** calls is executed, depending on which of the conditions is true. Each condition is tested in turn, and if one is true, the corresponding statement is executed, and the rest are skipped. If none of the conditions is true, we fall through to the last one (line 9), printing "F".

Listing 4-3: Nested If-Else to Choose between Alternatives

```
1. if(grade >= 90)
2.     printf("A");
3. else if(grade >= 80)
4.     printf("B");
5. else if(grade >= 70)
6.     printf("C");
7. else if(grade >= 60)
8.     printf("D");
9. else printf("F");
```

Another example of this arrangement, assume that a real variable x is known to be greater than or equal to zero and less than one. The following multiple choice decision increments `count1` if $0 \leq x < 0.25$, increments `count2` if $0.25 \leq x < 0.5$, increments `count3` if $0.5 \leq x < 0.75$ and increments `count4` if $0.75 \leq x < 1$.

```
if (x < 0.25)
    count1=count1+1;
else if (x < 0.5)
    count2=count2+1;
else if (x < 0.75)
    count3=count3+1;
else
    count4=count4+1;
```

Note how the ordering of the tests here has allowed the simplification of the conditions. For example when checking that x lies between 0.25 and 0.50 the test $x < 0.50$ is only carried out if the test $x < 0.25$ has already failed hence x is greater than 0.25. This shows that if x is less than 0.50 then x must be between 0.25 and 0.5. Compare the above with the following clumsy version using more complex conditions:

```
if (x < 0.25)
    count1=count1+1;
else if (x >= 0.25 && x < 0.5)
```

```

        count2=count2+1;
    else if (x >= 0.5 && x < 0.75)
        count3=count3+1;
    else
        count4=count4+1;

```

4.3 Switch Statement

In the last section, it was shown how a choice could be made using nested if-else statements. However a less unwieldy method in some cases is to use a **switch** statement. For example the **switch** statement used in Listing 4-4 sets the variable **grade** to the character A, B or C depending on whether the variable **i** has the value 1, 2, or 3.

Listing 4-4: Example of a Switch Statement

```

1. switch (i){
2.     case 1 : grade = 'A';
3.         break;
4.     case 2 : grade = 'B';
5.         break;
6.     case 3 : grade = 'c';
7.         break;
8.     default : printf("%d is not in range",i);
9.         break;
10. }

```

The general form of a **switch** statement is:

```

switch (selector){
    case label1: statement1;
        break;
    case label2: statement2;
        break;
    ...
    case labeln: statementn;
        break;
    default: statementd;           // optional
        break;
}

```



The **selector** may be an integer or character variable or an expression that evaluates to an integer or a character. The selector is evaluated and the value compared with each of the **case labels**. The case labels must have the same type as the selector and they must all be different. If a match is found between the selector and one of the case labels, say `labeli`, then the statements from the statement `statementi` **until the next break** statement will be executed. If the value of the selector cannot be matched with any of the case labels then the statement associated with **default** is executed. The default is optional but it should only be left out if it is certain that the selector will always take the value of one of the case labels. Note that the statement associated with a case label can be a single statement or a sequence of statements (without being enclosed in braces `{}`).

The following statement writes out the day of the week depending on the value of an integer variable `day`. It assumes that day 1 is Sunday.

```
switch (day){
    case 1 : printf("Sunday");
             break;
    case 2 : printf("Monday");
             break;
    case 3 : printf("Tuesday");
             break;
    case 4 : printf("Wednesday");
             break;
    case 5 : printf("Thursday");
             break;
    case 6 : printf("Friday");
             break;
    case 7 : printf("Saturday");
             break;
    default : printf("Not an allowable day number");
             break;
}
```

If it has already been ensured that `day` takes a value between 1 and 7 then the default case may be missed out.

It is allowable to associate several case labels with one statement. For example if the above example is amended to write out whether day is a weekday or is part of the weekend:

```
switch (day){
    case 1 :
    case 7 : printf("This is a weekend day");
            break;
    case 2 :
    case 3 :
    case 4 :
    case 5 :
    case 6 : printf("This is a weekday");
            break;
    default : printf("Not a legal day");
            break;
}
```

4.4 Boolean Expressions

An if statement like

```
if(x > max)

    max = x;
```

is perhaps deceptively simple. Conceptually, we say that it checks whether the condition `x > max` is “**true**” or “**false**”. Therefore, we need to understand how true and false values are represented, and how they are interpreted by statements like `if`.

In C/C++, a true/false¹ condition can be represented as an *integer*. In C, false is represented by a value of 0 (zero), and true is represented by any non-zero value. Since there are many non-zero values, when we have to pick a specific value for “true” we’ll pick 1 (one).

The **relational operators** such as `<`, `<=`, `>`, and `>=` are in fact operators, just like `+`, `-`, `*`, and `/`. The relational operators take two values,

¹ mathematics involving only two values is called Boolean Algebra after George Boole, a mathematician who refined this study.

look at them, and return a value of 1 or 0 depending on whether the tested relation was true or false. The complete set of relational operators in C is presented in Table 4-1.

Table 4-1: Relational Operators

Relational Operator	Function
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
==	Equal
!=	Not equal

For example, $1 < 2$ is 1, $3 > 4$ is 0, $5 == 5$ is 1, and $6 != 6$ is 0.



We've now encountered perhaps the most common error in C: the equality-testing operator is `==`, not a single `=`, which is assignment. If you accidentally write:

```
if(a = 0)
```

(and you probably will at some point; everybody makes this mistake), it will not test whether `a` is zero, as you probably intended. Instead, it will assign zero to `a`, and then check the “true” branch of the `if` statement (if `a` is a **non-zero**). But `a` will have just been assigned the value 0, so the “true” branch will never be taken! (This could drive you crazy while debugging--you wanted to do something if `a` was 0, and after the test, `a` is 0, whether it was supposed to be or not, but the “true” branch is nevertheless not taken.)



To avoid this problem, always start your relational expression with the number:

```
if(0 = a)
```

In case that you wrote a single equal, the compiler will through an error as an assignment expression can't start with a number.

The relational operators work with arbitrary numbers and generate true/false values. You can also combine true/false values by using the **Boolean operators**, which take true/false values as operands and compute new true/false values. The three Boolean operators are shown in Table 4-2.

Table 4-2: Boolean Operators

Boolean Operator	Function
&&	Logical AND
	Logical OR
!	Logical NOT (Unary operator; takes one operand)


The && (“AND”) operator takes two true/false values and produces a true (1) result if both operands are true (that is, if the left-hand side is true and the right-hand side is true). The || (“OR”) operator takes two true/false values and produces a true (1) result if either operand is true. The ! (“NOT”) operator takes a single true/false value and negates it, turning false to true and true to false (0 to 1 and nonzero to 0).

For example, to test whether the variable `i` lies between 1 and 10, you might use

```
if(1 < i && i < 10)
    ...
```

Here we're expressing the relation “`i` is between 1 and 10”. It's important to understand why the more obvious expression:

```
if(1 < i < 10)                /* WRONG */
```

 would not work. The expression `1 < i < 10` is parsed by the compiler analogously to `1 + i + 10`. The expression `1 + i + 10` is parsed as `(1 + i) + 10` and means “add 1 to `i`, and then add the result

to 10". Similarly, the expression `1 < i < 10` is parsed as `(1 < i) < 10` and means "see if 1 is less than `i`, and then see if the result is less than 10". But in this case, "the result" is 1 or 0, depending on whether `i` is greater than 1. Since both 0 and 1 are less than 10, the expression `1 < i < 10` would always be true in C, regardless of the value of `i`!

As far as C is concerned, the controlling expression (of conditional statements like `if`) can in fact be any expression: it doesn't have to "look like" a Boolean expression; it doesn't have to contain relational or logical operators. C just looks at whether the expression evaluates to 0 or non-zero. For example, if you have a variable `x`, and you want to do something if `x` is non-zero, it's possible to write:

```
if(x)
    statement
```

and the statement will be executed if `x` is non-zero (since non-zero means "true").

4.5 While Loops



Loops generally consist of two parts: one or more **control expressions** that control the execution of the loop, and the **body**, which is the statement or set of statements which is executed over and over.

The most basic loop in C is the **while** loop. A **while** loop has one control expression, and executes as long as that expression is true. Example in Listing 4-5 repeatedly doubles the number 2 (2, 4, 8, 16, ...) and prints the resulting numbers as long as they are less than 1000. As we can see in the example, we have used braces `{}` to enclose the group of statements which are to be executed together as the body of the loop.

The general syntax of a while loop is

```
while( expression )
    statement
```

Listing 4-5: Example of a While Loop

```
1. int x = 2;
2. while(x < 1000){
3.     printf("%d\n", x);
4.     x = x * 2;           //very important
5. }
```



A while loop starts out like an `if` statement; if the condition expressed by the expression is true, the statement is executed. However, after executing the statement, the condition is tested again, and if it's still true, the statement is executed again. (***Presumably, the condition depends on some value which is changed in the body of the loop.***) As long as the condition remains true, the body of the loop is executed over and over again. If the condition is false right at the start, the body of the loop is never executed.

As another example, if you want to print a number of blank lines, with the variable `n` holding the number of blank lines to be printed, you might use code like this:

```
while(n > 0){
    printf("\n");
    n = n - 1;
}
```

After the loop finishes, `n` will have the value 0.

You use a `while` loop when you have a statement or group of statements which may have to be executed a number of times to complete their task. The controlling expression represents the ***condition***. There's more work to do as long as the expression is true, the ***body*** of the loop is executed; presumably, it makes at least some progress at its task. When the expression becomes false, the task is done, and the rest of the program (beyond the loop) can proceed. When we think about a loop in this way, we can see an additional important property: if the expression evaluates to false before the very first trip through the loop, we make zero trips through the loop. In other words, if the task is already done (if there's no

work to do) the body of the loop is not executed at all. It's always a good idea to think about the “**boundary conditions**” in a piece of code, and to make sure that the code will work correctly when there is no work to do, or when there is a trivial task to do, such as sorting an array of one number. Experience has shown that bugs at boundary conditions are quite common.

4.6 For Loops

Our second loop, which we've seen at least one example of already, is the **for** loop. The first one we saw was:

```
for (i = 0; i < 10; i = i + 1)
    printf("i is %d\n", i);
```

More generally, the syntax of a for loop is

```
for( expr1; expr2 ; expr3)
    statement
```

Here we see that the for loop has three control expressions. As always, the statement can be a brace-enclosed block.

Many loops are set up to cause some variable to step through a range of values, or, more generally, to set up an initial condition and then modify some value to perform each succeeding loop as long as some condition is true. The three expressions in a for loop encapsulate these conditions: **expr₁** sets up the **initial condition**, **expr₂** **tests** whether another trip through the loop should be taken, and **expr₃** **increments** or updates things after each trip through the loop and prior to the next one. In our first example, we had **i = 0** as **expr₁**, **i < 10** as **expr₂**, **i = i + 1** as **expr₃**, and the call to **printf** as the body statement of the loop. So the loop began by setting **i** to 0, proceeded as long as **i** was less than 10, printed out **i**'s value during each trip through the loop, and added 1 to **i** between each trip through the loop.

When the compiler sees a **for** loop, first, **expr₁** is executed. Then, **expr₂** is evaluated, and if it is true, the body of the loop (statement) is executed. Then, **expr₃** is executed to go to the next step, and **expr₂** is evaluated again, to see if there is a next step. During the execution of a **for** loop, the sequence is:

```

expr1
expr2
    statement
expr3
expr2
    statement
expr3
...
expr2
    statement
expr3
expr2

```

The first thing executed is `expr1`. `expr3` is executed after every trip through the loop. The last thing executed is always `expr2`, because when `expr2` evaluates false, the loop exits.

All three expressions of a **for** loop are optional. If you leave out `expr1`, there is simply no initialization step, and the variable(s) used with the loop had better have been initialized already. If you leave out `expr2`, there is no test, and the default for the **for** loop is that another trip through the loop should be taken (such that unless you break out of it some other way, the loop runs forever). If you leave out `expr3`, there is no increment step.

The semicolons separate the three controlling expressions of a **for** loop. If you leave out one or more of the expressions, the semicolons **remain**. Therefore, one way of writing a deliberately infinite loop in C is

```
for(;;)
```

```
...
```

It's also worth noting that a **for** loop can be used in more general ways than the simple, iterative examples we've seen so far. The “**control variable**” of a **for** loop does not have to be an integer, and it does not have to be incremented by an additive increment. It could be

“incremented” by a multiplicative factor (1, 2, 4, 8, ...) if that was what you needed, or it could be a floating-point variable, or it could be another type of variable which we haven't met yet which would step, not over numeric values, but over the elements of an array or other data structure.

The powers-of-two example of the previous section does fit this pattern, so we could rewrite it like this:

```
int x;

for(x = 2; x < 1000; x = x * 2)

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

There is no earth-shaking or fundamental difference between the **while** and **for** loops. In fact, given the general **for** loop

```
for( expr1; expr2 ; expr3)

    statement
```

you could usually rewrite it as a **while** loop, moving the initialize and increment expressions to statements before and within the loop:

```
expr1;

while(expr2){

    statement

    expr3 ;

}
```

An important contrast between the **for** and **while** loops is that although the test expression (**expr₂**) is optional in a **for** loop, it is **required** in a **while** loop. If you leave out the controlling expression of a **while** loop, the compiler will complain about a **syntax error**. To write a deliberately infinite while loop, you have to supply an expression which is always non-zero. The most obvious one would simply be **while(1)** .

If it's possible to rewrite a **for** loop as a **while** loop and vice versa, why do they both exist? Which one should you choose? In general, when you choose a **for** loop, its three expressions should all manipulate the same variable or data structure, using the initialize, test, increment pattern. If they don't manipulate the same variable or don't follow that pattern,

wedging them into a **for** loop buys nothing and a **while** loop would probably be clearer. The reason that one loop or the other can be clearer is simply that, when you see a **for** loop, you expect to see an idiomatic initialize/test/increment of a single variable, and if the **for** loop you're looking at doesn't end up matching that pattern, you've been momentarily misled.

4.7 Break and Continue

Sometimes, due to an exceptional condition, you need to jump out of a loop early, that is, before the main controlling expression of the loop causes it to terminate normally. Other times, in an elaborate loop, you may want to jump back to the top of the loop (to test the controlling expression again, and perhaps begin a new trip through the loop) without playing out all the steps of the current loop. The **break** and **continue** statements allow you to do these two goals.

To put everything we've seen in this chapter together, as well as demonstrate the use of the **break** statement,

Listing 4-6 shows a program for printing prime numbers between 1 and 100.

The outer loop steps the variable **i** through the numbers from 3 to 100; the code tests to see if each number has any divisors other than 1 and itself. The trial divisor **j** loops from 2 up to **i**. **j** is a divisor of **i** if the remainder of **i** divided by **j** is 0, so the code uses C's “remainder” or “modulus” operator **%** to make this test.

If the program finds a divisor, it uses **break** to break out of the inner loop, without printing anything. But if it notices that **j** has risen higher than the square root of **i**, without its having found any divisors, then **i** must not have any divisors, so **i** is prime, and its value is printed. (Once we've determined that **i** is prime by noticing that **j > sqrt(i)**, there's no need to try the other trial divisors, so we use a second **break** statement to break out of the loop in that case, too.)

Listing 4-6: Printing Prime Numbers between 1 and 100

```

1. #include <stdio.h>
2. #include <math.h>
3. int main(){
4.     int i, j;
5.     printf("%d\n", 2);

6.     for(i = 3; i <= 100; i = i + 1){
7.         for(j = 2; j < i; j = j + 1){

8.             if(i % j == 0)
9.                 break;

10.            if(j > sqrt(i)){
11.                printf("%d\n", i);
12.                break;
13.            }
14.        }
15.    }
16.    return 0;
17. }

```

The simple algorithm and implementation we used here does not work for 2, the only even prime number, so the program “cheats” and prints out 2 no matter what, before going on to test the numbers from 3 to 100.



Many improvements to this simple program are of course possible; you might experiment with it. Did you notice that the “test” expression of the inner loop `for(j = 2; j < i; j = j + 1)` is in a sense unnecessary, because the loop always terminates early due to one of the two `break` statements?

We end up this chapter with a simple example of a **continue** statement. In this example we print the numbers from 1 to 5 and skip number 3 using the **continue** statement.

```

for (j = 1; j <= 5; j=j+1) {
    if (j == 3) {
        printf("continue!");
        continue;
    }
    printf("%d\n",j);
}

```

}

The output of this program should be as follows:

1

2

Continue!

4

5

4.8 Exercises

1. What value is returned by the following code fragment?

```
int i = 7;
y = y+i;
if(i < 8)
    printf("The value is less than eight.\n");
```

- a) 7
- b) The value is less than eight.
- c) 8
- d) No value is returned, since the test is false.

2. Which of the following is the correct nested if code?

- a)

```
if(c >= '0')
    if(c <= '9');
        printf("This is a number.\n");
```
- b)

```
if(c >= '0');
    if(c <= '9');
        printf("This is a number.\n");
```
- c)

```
if(c >= '0')
    if(c <= '9')
        printf("This is a number.\n");
```
- d)

```
if(c >= '0'):
    if(c <= '9')
        printf("This is a number.\n");
```

3. Which of the following is a correct code fragment from a case construct?

- a) case 1;
 rate = .01;
 break;
- b) case 2:
 rate = .02;
 break;
- c) case 3
 rate = .03;
 break;
- d) case 4:
 rate = .01
 break;

4. Which of the following identifies the purpose of 'default' in a switch statement?
- a) it terminates the switch statement
 - b) it identifies the values being compared
 - c) it executes only if the test value does not equal any of the other cases in the switch
 - d) it causes the compiler to skip the switch statement
5. Which one of the following code fragments is written correctly?
- a) for(x = 0, x <= 200, x=x+1)
 - b) for(x = 0; x <= 200; x=x+1);
 - c) for(x = 0; x <= 200; x=x+1)
 - d) for(x = 0; x <= 200; x=x+1;)

6. How many times will the following message be printed?

```
for ( x = 10, y = 0; x < 100; x =x+ 10, y=y+1)
    printf("this is a test \n");
```

- a) 1 time
- b) 9 times
- c) 10 times
- d) 100 times

7. In writing a for loop, which code fragment would be accepted?

- a) `for(; x <= 10; x=x+1;)`
- b) `for(; x <= 10; x=x+1)`
- c) `for(x <= 10; x=x+1)`
- d) `for(x <= 10; ; x=x+1)`

8. Which of the following code fragments is correct?

- a)

```
while{
    (x < 21)
    printf("Hit me again\n");
    x=x+1;
}
```
- b)

```
while (x < 21);
{
    printf("Hit me again\n");
    x=x+1;
}
```
- c)

```
while (x < 21) x=x+1;
{
    printf("Hit me again\n");
}
```
- d)

```
while (x < 21){
    printf("Hit me again\n");
    x=x+1;
}
```

9. At a minimum, how many times will the loop body of a

- while loop be executed?
- a) less than one time
 - b) one time
 - c) two times
 - d) more than two times
10. With a while loop, what loop control component is required?
- a) initial value
 - b) test condition
 - c) loop increment
 - d) do
11. Which of the following code fragments is correct?
- a)

```
do
{
    while (x < 21)
        printf("Hit me again\n");
}
x=x+1;
```
 - b)

```
do-while
{
    printf("Hit me again\n");
    x=x+1;
}
```
 - c)

```
do
{
    printf("Hit me again\n");
    x=x+1;
}while(x < 21);
```
 - d)

```
do
{
    printf("Hit me again\n");
    x=x+1;
}while (x < 21)
```
12. With a do-while loop, which of the following is

executed first?

- a) loop body
- b) while statement
- c) test condition
- d) loop control

13. With a do-while loop, the loop body is executed if the test condition is

- a) true, but not false
- b) false, but not true
- c) either true or false
- d) false to begin with and true later on

14. At a minimum, how many times will the loop body of a do-while loop be executed?

- a) less than one time
- b) one time
- c) two times
- d) more than two times

15. Which one of the following does not terminate the execution of a loop?

- a) continue statement
- b) break statement
- c) goto statement
- d) All of the above terminate the execution of a loop.

16. Given the following code fragment, what values will be printed when $x = 4$?

```

for ( x = 1; x <= 5; x=x+1)
{
    y = 1;
    while (y <= 3)
    {
        printf("%3d",x*y);
        y++;
    }
}

```

- a) 1 2 3
- b) 5 6 7
- c) 4 8 12
- d) 5 10 15

17. What would the equivalent code, using a while loop, be for the example

```

for(i = 0; i < 10; i = i + 1)
    printf("i is %d\n", i);

```

18. What is the numeric value of the expression $3 < 4$?

19. Under what conditions will this code print "water"?

```

if(T < 32)
    printf("ice\n");
else if(T < 212)
    printf("water\n");
else
    printf("steam\n");

```

20. What would this code print?

```

int x = 3;
if(x)
    printf("yes\n");
else
    printf("no\n");

```

21. (trick question) What would this code print?

```

int i;
for(i = 0; i < 3; i = i + 1)
    printf("a\n");
    printf("b\n");
printf("c\n");

```

22. Write a program to find out how many of the numbers from 1 to 10 are greater than 3. (The answer, of course,

should be 7.) Your program should have a loop which steps a variable over the 10 numbers, after the loop has finished, print out the count).

23. Write a program to compute the average of the ten numbers 1, 4, 9, ..., 81, 100, that is, the average of the squares of the numbers from 1 to 10.

Note: you should keep track of the sum in a variable of type float or double to get the answer as a floating-point number, which you should print out using %f in the printf format string, not %d. (In a printf format string, %d prints only integers, and %f is one way to print floating-point numbers. In this case, the answer should be 38.5

24. Write a program to print the numbers between 1 and 10, along with an indication of whether each is even or odd, like this:

```
1 is odd
2 is even
3 is odd
...
```

(Hint: use the % operator.)

25. Write a program to print the first 10 Fibonacci numbers. Each Fibonacci number is the sum of the two preceding ones. The sequence starts out 0, 1, 1, 2, 3, 5, 8, ...

26. Write a program to print this triangle:

```
*
**
***
****
*****
*****
*****
*****
*****
*****
*****
```

Don't use ten printf statements; use two nested loops instead. You'll have to use braces around the body of the outer loop if it contains multiple statements:

```
for(i = 1; i <= 10; i = i + 1){
    /* statements */
}
```



5 More about Declarations and Operators

In this Chapter we will consider declarations and usage of a very important concept in C called arrays. At the same time we will consider more advanced arithmetic operators in C.

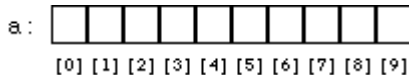
5.1 Arrays

The concept of an array is common to most programming languages. In an array, multiple values of the same data type can be stored with one variable name. The use of arrays allows for the development of smaller and more clearly readable programs.

To declare an array of several elements:

```
int a[10];
```

declares an array, named **a**, consisting of ten elements, each of type **int**. Simply speaking, an array is a variable that can hold more than one value. You specify which of the several values you're referring to at any given time by using a **numeric index**. (Arrays in programming are similar to vectors or matrices in mathematics) We can represent the array **a** above with a picture like this:



In C, arrays are zero-based: the ten elements of a 10-element array are numbered from 0 to 9. The index that specifies a single element of an array is simply an **integer** expression in square brackets. The first element of the array is **a[0]**, the second element is **a[1]**, etc. You can use these “array index expressions” anywhere you can use the name of a simple variable, for example:

```
a[0] = 10;  
a[1] = 20;  
a[2] = a[0] + a[1];
```

Notice that the indexed array references (i.e. expressions such as `a[0]` and `a[1]`) can appear on either side of the assignment operator.

The index does not have to be a constant like 0 or 1; it can be any integral expression. For example, it's common to loop over all elements of an array:

```
int i;
for(i = 0; i < 10; i = i + 1)
    a[i] = 0;
```

This loop sets all ten elements of the array `a` to 0.



Arrays are a real convenience for many problems, but there is not a lot that C will do with them for you automatically. In particular, you can neither set all elements of an array at once nor assign one array to another; both of the assignments

```
a = 0;                                /* WRONG */
and
int b[10];
b = a;                                /* WRONG */
```

are illegal.

To set all of the elements of an array to some value, you must do so one by one, as in the loop example above. To copy the contents of one array to another, you must again do so one by one:

```
int b[10];

for(i = 0; i < 10; i = i + 1)
    b[i] = a[i];
```



Remember that for an array declared:

```
int a[10];
```

there is no element `a[10]`; the topmost element is `a[9]`. This is one reason that zero-based loops are also common in C. Note that the for loop

```
for(i = 0; i < 10; i = i + 1)
    ...
```

does just what you want in this case: it starts at 0, the number 10 suggests (correctly) that it goes through 10 iterations, but the less-than

comparison means that the last trip through the loop has `i` set to 9. (The comparison `i <= 9` would also work, but it would be less clear and therefore poorer style.)

In the little examples so far, we've always looped over all 10 elements of the sample array `a`. It's common, however, to use an array that's bigger than necessarily needed, and to use a second variable to keep track of how many elements of the array are currently in use. For example, we might have an integer variable

```
int numElements;
```

Then, when we wanted to do something with array `a` (such as print it out), the loop would run from 0 to `numElements`, not 10 (or whatever `a`'s size was):

```
for(i = 0; i < numElements; i = i + 1)
    printf("%d\n", a[i]);
```

Naturally, we would have to ensure ensure that `numElements` value was always less than or equal to the number of elements actually declared in `a`. Arrays are not limited to type `int`; you can have arrays of `char` or `double` or any other type.

The code in

Listing 5-1 is a slightly larger example of the use of arrays. Suppose we want to investigate the behavior of rolling a pair of dice. The total roll can be anywhere from 2 to 12, and we want to count how often each roll comes up. We will use an array to keep track of the counts: `a[2]` will count how many times we've rolled 2, etc.

We'll simulate the roll of a die by calling C's ***random number generation function***, `rand()`. Each time you call `rand()`, it returns a different, pseudo-random integer. The values that `rand()` returns typically span a large range, so we'll use C's modulus (or "remainder") operator `%` to produce random numbers in the range we want. The expression `rand() % 6` produces random numbers in the range 0 to 5, and `rand() % 6 + 1` produces random numbers in the range 1 to 6.

Listing 5-1: Example of an Array Usage

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

3. int main(){
4.     int i;
5.     int d1, d2;
6.     int a[13];          /* uses [2..12] */
7.
8.     for(i = 2; i <= 12; i = i + 1)
9.         a[i] = 0;

10.    for(i = 0; i < 100; i = i + 1){
11.        d1 = rand() % 6 + 1;
12.        d2 = rand() % 6 + 1;
13.        a[d1 + d2] = a[d1 + d2] + 1;
14.    }

15.    for(i = 2; i <= 12; i = i + 1)
16.        printf("%d: %d\n", i, a[i]);

17.    return 0;
18. }
```

We include the header `<stdlib.h>` because it contains the necessary declarations for the `rand()` function. We declare the array of size 13 so that its highest element will be `a[12]`. (We're wasting `a[0]` and `a[1]`; this is no great loss.) The variables `d1` and `d2` contain the rolls of the two individual dice; we add them together to decide which cell of the array to increment, in the line

`a[d1 + d2] = a[d1 + d2] + 1;`

After 100 rolls, we print the array out.

5.1.1 Array Initialization

It is possible to initialize some or all elements of an array when the array is defined. The syntax looks like this:

```
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

The list of values, enclosed in braces {}, separated by commas, provides the initial values for successive elements of the array.

If there are fewer initializers than elements in the array, the remaining elements are automatically initialized to 0. For example,

```
int a[10] = {0, 1, 2, 3, 4, 5, 6};
```

would initialize `a[7]`, `a[8]`, and `a[9]` to 0. When an array definition includes an initializer, the array dimension may be omitted, and the compiler will infer the dimension from the number of initializers. For example,

```
int b[] = {10, 11, 12, 13, 14};
```



would declare, define, and initialize an array `b` of 5 elements (i.e. just as if you'd typed `int b[5]`). Only the dimension is omitted; the brackets [] remain to indicate that `b` is in fact an array.

In the case of arrays of `char`, the initializer may be a string constant:

```
char s1[7] = "Hello,";  
char s2[10] = "there,";  
char s3[] = "world!";
```

As before, if the dimension is omitted, it is inferred from the size of the string initializer. (We haven't covered strings in detail yet--we'll do so in a later chapter--but it turns out that all strings in C are terminated by a **special character with the value 0** (null character). Therefore, the array `s3` will be of size 7, and the explicitly-sized `s1` does need to be of size at least 7. For `s2`, the last 4 characters in the array will all end up being this zero-value character.)

5.1.2 Arrays of Arrays (Multidimensional Arrays)

When we said that "Arrays are not limited to type `int`; you can have arrays of... any other type," we meant that more literally than you might

have guessed. If you have an “array of `int`” it means that you have an array each of whose elements is of type `int`. But you can have an array each of whose elements is of type `x`, where `x` is any type you choose. In particular, you can have an array each of whose elements is another array! We can use these arrays of arrays for the same sorts of tasks as we would use multidimensional arrays in matrices in mathematics. Naturally, we are not limited to arrays of arrays, either; we could have an array of arrays of arrays, which would act like a 3-dimensional array, etc.

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Figure 5-1: Multidimensional Array

The declaration of an array of arrays looks like this:

```
int a2[5][7];
```

You have to read complicated declarations like these ***inside out***. What this one says is that `a2` is an array of 5 somethings, and that each of the somethings is an array of 7 ints. More briefly, “`a2` is an array of 5 arrays of 7 ints”. You can think of `a2` as having 5 ***rows*** and 7 ***columns***.

To illustrate the use of multidimensional arrays, we might fill in the elements of the above array `a2` using this piece of code:

```
int i, j;
for(i = 0; i < 5; i = i + 1){           //iterate on rows
    for(j = 0; j < 7; j = j + 1) // iterate on columns
        a2[i][j] = 10 * i + j;
}
```

This pair of nested loops sets `a[1][2]` to 12, `a[4][1]` to 41, etc. Since the first dimension of `a2` is 5, the first index variable, `i`, runs from 0 to 4. Similarly, the second index varies from 0 to 6.

We could print **a2** out (in a two-dimensional way, suggesting its structure) with a similar pair of nested loops:

```
for(i = 0; i < 5; i = i + 1){
    for(j = 0; j < 7; j = j + 1)
        printf("%d\t", a2[i][j]);
    printf("\n");
}
```

(The character `\t` in the `printf` string is the tab character.)

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {
    {0, 1, 2, 3} , /* initializers for row indexed by 0 */
    {4, 5, 6, 7} , /* initializers for row indexed by 1 */
    {8, 9, 10, 11} /* initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

5.2 More Operators

It's extremely common in programming to have to increment a variable by 1, that is, to add 1 to it. (For example, if you're processing each element of an array, you'll typically write a loop with an index variable stepping through the elements of the array, and you'll increment the variable each time through the loop.) The classic way to increment a variable is with an assignment like:

```
i = i + 1
```

Such an assignment is perfectly common and acceptable, but it has a few slight problems:

- a. It looks a little odd, especially from an algebraic perspective.

- b. If the object being incremented is not a simple variable, the idiom can become cumbersome to type, and correspondingly more error-prone. For example, the expression

$$a[i+j+2*k] = a[i+j+2*k] + 1$$

is a bit of a mess, and you may have to look closely to see that the similar-looking expression

$$a[i+j+2*k] = a[i+j+2+k] + 1$$

probably has a mistake in it.

Since incrementing things is so common, it might be nice to have an easier way of doing it. In fact, C provides not one but two other, simpler ways of incrementing variables and performing other similar operations.

5.2.1 Assignment Operators

The first and more general way is that any time you have the pattern

$$v = v \text{ op } e$$

where v is any variable (or anything like $a[i]$), op is any of the binary arithmetic operators we've seen so far, and e is any expression, you can replace it with the simplified

$$v \text{ op} = e$$

For example, you can replace the expressions

$$i = i + 1$$
$$j = j - 10$$
$$k = k * (n + 1)$$
$$a[i] = a[i] / b$$

with

$$i += 1$$

```
j -= 10
```

```
k *= n + 1
```

```
a[i] /= b
```

In a previous example in this chapter, we have used the assignment:

```
a[d1 + d2] = a[d1 + d2] + 1;
```

to count the rolls of a pair of dice. Using `+=`, we could simplify this expression to

```
a[d1 + d2] += 1;
```

As these examples show, you can use the “`op=`” form with any of the arithmetic operators. The expression, `e`, does not have to be the constant 1; it can be any expression. You don't always need as many explicit parentheses when using the `op=` operators: the expression

```
k *= n + 1;
```

is interpreted as

```
k = k * (n + 1);
```

5.2.2 Increment and Decrement Operators

The assignment operators of the previous section let us replace `v = v op e` with `v op= e`, so that we didn't have to mention `v` twice. In the most common cases, namely when we're adding or subtracting the constant 1 (that is, when `op` is `+` or `-` and `e` is 1), C provides another set of shortcuts: the ***auto-increment and auto-decrement operators***. In their simplest forms, they look like this:

```
++i          add 1 to i
```

```
--j          subtract 1 from j
```

These correspond to the slightly longer `i += 1` and `j -= 1` respectively.

The `++` and `--` operators apply to one operand (they're **unary operators**). The expression `++i` adds 1 to `i`, and stores the incremented result back in `i`. This means that these operators don't just compute new values; they also modify the value of some variable.

The incremented (or decremented) result is also made available to the rest of the expression, so an expression like:

```
k = 2 * ++i;
```

means “add one to `i`, store the result back in `i`, multiply it by 2, and store that result in `k`”. So for example if `i` here equals 1, after the execution of this expression, `i` will equal 2 and `k` will equal 4.



Both the `++` and `--` operators have an unusual property: they can be used in two ways, depending on whether they are written to the left or the right of the variable they are operating on. In either case, they increment or decrement the variable they are applied to; the difference concerns whether it's the old or the new value that's “returned” to the surrounding expression. The **prefix** form `++i` increments `i` and returns the incremented value. The **postfix** form `i++` increments `i`, but returns the old, non-incremented value. Rewriting our previous example slightly, the expression:

```
k = 2 * i++;
```



means “take `i`'s old value and multiply it by 2, store the result of the multiplication in `k`, and then increment `i`”. So if `i` equals 1, after the execution of this expression, `i` will equal 2 and `k` will equal 2.

The distinction between the prefix and postfix forms of `++` and `--` will probably seem strained at first, but it will make more sense once we begin using these operators in more realistic situations.

As a simple example, consider this code fragment:

```
int i=1;
printf("i is %d\n", i++);           // i is 1
printf("i is %d\n", ++i);           // i is 3
```


In the first `printf` statement, we are using the postfix form that returns first the current value of `i` to the `printf` statement and then increment `i` to 2. In the second `printf` the prefix form increments `i` first to 3, then returns its new value to the `printf` statement.



Notice that the `++` operator doesn't just mean “add one”; it means “add one to a variable” or “make a variable's value one more than it was before”. So try to avoid which some confused programmers sometimes write, presumably because they want to be extra sure that `i` is incremented by 1:

```
i = i++;
```

But `i++` all by itself is sufficient to increment `i` by 1; the extra (explicit) assignment to `i` is unnecessary and in fact counterproductive, meaningless, and incorrect. If you want to increment `i` (that is, add one to it, and store the result back in `i`), either use:

```
i = i + 1;
```

or

```
i += 1;
```

or

```
++i;
```

or

```
i++;
```

Did it matter whether we used `++i` or `i++` in this last example? Remember, the difference between the two forms is what value (either the old or the new) is passed on to the surrounding expression. If there is no surrounding expression, if the `++i` or `i++` appears all by itself, to increment `i` and do nothing else, you can use either form; it makes no difference. Two ways that an expression can appear “all by itself”. When it is an expression statement terminated by a semicolon, as above, or

when it is one of the controlling expressions of a for loop. For example, both the loops

```
for(i = 0; i < 10; ++i)

    printf("%d\n", i);
```

and

```
for(i = 0; i < 10; i++)

    printf("%d\n", i);
```

will behave exactly the same way and produce exactly the same results.

In the preceding section, we simplified the expression:

```
a[d1 + d2] = a[d1 + d2] + 1;
```

from a previous chapter down to

```
a[d1 + d2] += 1;
```

Using ++, we could simplify it still further to

```
a[d1 + d2]++;
```

or

```
++a[d1 + d2];
```

(Again, in this case, both are equivalent.)

5.2.3 Order of Evaluation

When you start using the ++ and -- operators in larger expressions, you end up with expressions which do several things at once, i.e., they modify several different variables at more or less the same time. When you write such an expression, you must be careful not to have the expression to assign two different values to the same variable, or by assigning a new value to a variable at the same time that another part of the expression is trying to use the value of that variable.

Assume for example the following expression:

```
a[i++] = b[j++];
```

which assigns `b[j]` to `a[i]`, and increments `i`, and increments `j`. If you're not careful, though, it's easy for this sort of thing to get out of hand. Can you figure out exactly what the expression:

```
a[i++] = b[i++];          /* WRONG */
```

should do? I can't, and here's the important part: ***neither can the compiler***. We know that the definition of postfix `++` is that the former value, before the increment, is what goes on to participate in the rest of the expression, but the expression `a[i++] = b[i++]` contains two `++` operators. Which of them happens first? Does this expression assign the old i^{th} element of `b` to the new i^{th} element of `a`, or vice versa? No one knows.

When the order of evaluation matters but is not well-defined (that is, when we can't say for sure which order the compiler will evaluate the various dependent parts in) we say that the meaning of the expression is undefined, and if we're smart we won't write the expression in the first place. Why would anyone ever write an “undefined” expression? Because sometimes, the compiler happens to evaluate it in the order a programmer wanted, and the programmer assumes that since it works, it must be okay.

For example, suppose we carelessly wrote this loop:

```
int i, a[10];
i = 0;
while(i < 10)
    a[i] = i++;          /* WRONG */
```



It looks like we're trying to set `a[0]` to 0, `a[1]` to 1, etc. But what if the increment `i++` happens before the compiler decides which cell of the array `a` to store the (un-incremented) result in? We might end up setting `a[1]` to 0, `a[2]` to 1, etc., instead. Since, in this case, we can't be

sure which order things would happen in, we simply shouldn't write code like this. In this case, what we're doing matches the pattern of a **for** loop, anyway, which would be a better choice:

```
for(i = 0; i < 10; i++)  
    a[i] = i;
```

Now that the increment **i++** isn't crammed into the same expression that's setting **a[i]**, the code is perfectly well-defined, and is guaranteed to do what we want.

In general, you should be wary of ever trying to second-guess the order an expression will be evaluated in, with two exceptions:



Although we haven't mentioned it yet, it is guaranteed that the logical operators **&&** and **||** are evaluated left-to-right, and that the **right-hand side is not evaluated** at all if the left-hand side determines the outcome.

To look at one more example, it might seem that the code:

```
int i = 7;  
printf("%d\n", i++ * i++);
```

would have to print 56, because no matter which order the increments happen in, $7*8$ is $8*7$ is 56. But **++** just says that the increment happens later, not that it happens immediately, so this code could print 49 (if the compiler chose to perform the multiplication first, and both increments later). And, it turns out that ambiguous expressions like this are such a bad idea that the ANSI C Standard does not require compilers to do anything reasonable with them at all. Theoretically, the above code could end up printing 42, or 8923409342, or 0, or crashing your computer.

Programmers sometimes mistakenly imagine that they can write an expression which tries to do too much at once and then predict exactly how it will behave based on “order of evaluation”. For example, we know that multiplication has higher precedence than addition, which means that in the expression

`i + j * k`

`j` will be multiplied by `k`, and then `i` will be added to the result. Informally, we often say that the multiplication happens before the addition. That's true in this case, but it doesn't say as much as we might think about a more complicated expression, such as

`i++ + j++ * k++`

In this case, besides the addition and multiplication, `i`, `j`, and `k` are all being incremented. We cannot say which of them will be incremented first; it's the compiler's choice. (In particular, it is not necessarily the case that `j++` or `k++` will happen first; the compiler might choose to save `i`'s value somewhere and increment `i` first, even though it will have to keep the old value around until after it has done the multiplication.)

In the preceding example, it probably doesn't matter -which variable is incremented first. It's not too hard, though, to write an expression where it does matter. In fact, we've seen one already: the ambiguous assignment `a[i++] = b[i++]`. We still don't know which `i++` happens first. (We cannot assume, based on the right-to-left behavior of the `=` operator, that the right-hand `i++` will happen first.) But if we had to know what `a[i++] = b[i++]` really did, we'd have to know which `i++` happened first.

Finally, note that parentheses don't dictate overall evaluation order any more than precedence does. Parentheses override precedence and say which operands go with which operators, and they therefore affect the overall meaning of an expression, but they don't say anything about the order of sub-expressions or side effects. We could not “fix” the evaluation order of any of the expressions we've been discussing by adding parentheses. If we wrote

`i++ + (j++ * k++)`

we still wouldn't know which of the increments would happen first. (The parentheses would force the multiplication to happen before the addition, but precedence already would have forced that, anyway.) If we wrote

`(i++) * (i++)`

the parentheses wouldn't force the increments to happen before the multiplication or in any well-defined order; this parenthesized version would be just as undefined as `i++ * i++` was.

Many programmers encourage you to write small programs to find out how your compiler implements some of these ambiguous expressions, but it's just one step from writing a small program to find out, to writing a real program which makes use of what you've just learned. But you don't want to write programs that work only under ***one particular compiler***, that take advantage of the way that one compiler (but perhaps no other) happens to implement the undefined expressions.



Please keep very firmly in mind that, for real programs, the very easiest way of dealing with ambiguous, undefined expressions (which one compiler interprets one way and another interprets another way and a third crashes on) is not to write them in the first place.

5.3 Exercises

1) Which data type is used for array subscripts?

- a) char only
- b) int only
- c) char and int only
- d) int, float, and char only

2) Given the following code fragment, what is the value of `arg1[5]`?

```
int arg1[] = {1,2,3,4,5};
```

- a) 0
- b) 4
- c) 5
- d) Not a meaningful value.

- 3) Given the following code fragment, which of the following is correct?

```
num[3] = 9;  
--num[3];
```

- a) num[3] = 9
 - b) num[2] = 9
 - c) num[3] = 8**
 - d) num[2] = 8
- 4) Which of the following declares a float array called worksheet[], with 30 rows and 50 columns?

- a) float worksheet array[30][50];
- b) float worksheet[50][30];
- c) float worksheet[30][50];
- e) worksheet[30][50] = float;

- 5) The shorthand expression for $x = x + 10$ is:

- a) $x += 10;$
- b) $+x = 10;$
- c) $x =+ 10;$
- d) $x = 10+;$

- 6) The result of the following arithmetic expression is?

```
y = 6 * 4 % 3 * 5;
```

- a) 0
- b) 9
- c) 30
- d) 40

- 7) What value does the computer give to the following expression if x is -4?

`((x <= 5) && (x != 0) && (x >= -5))`

- a) 0
- b) 1
- c) 2
- d) 3

- 8) Evaluate the following expressions that use the arithmetic operators where i = 10, j = 2, and k = 3.

Expression	Result
-----	-----
i - j * k	
i - i / j	
k % i + j	
i % 2 * k	
i + j * k - i / j	

- 9) Evaluate the following logical expressions where i = 10, j = 2, and k = 3.

Expression	Result
-----	-----
i - 2 * (j + k) i	
i - 2 * (j + k) && i	
j = i k	
j = i k j	
j - k j && k	

10) Evaluate following expressions where i = 10, j = 2, and k = 3.

Expression	Result
i += j	
i = j + k--	
i=j + k++	
i=j-- + ++k	

11. What's wrong with this scrap of code?

```
int a[5];
for(i = 1; i <= 5; i = i + 1)
    a[i] = 0;
```

12. What is the difference between the prefix and postfix forms of the ++ operator?

13. (trick question) What would the expression

```
i = i++
do?
```

14. Write a program to perform the following:

- Define and initialize an array with the following data

6	-2	8	-13	2	7
---	----	---	-----	---	---
- Calculate and print to screen the total and average of the data
- Calculate and print to screen the maximum value
- Calculate and print to screen the minimum value

15. Write a program to sort and print out the array in the previous question.

16. Write a program to define and initialize the following two matrices and then find the summation of them in a third matrix.

$$\begin{bmatrix} 100 & 13 & 2 \\ -5 & 4 & 6 \\ 12 & 11 & 10 \end{bmatrix} \quad \begin{bmatrix} 9 & -3 & 12 \\ 5 & 10 & 44 \\ 9 & -9 & 0 \end{bmatrix}$$



6 Functions and Program Structure

A **function** is a “black box” that we’ve locked part of our program into. The idea behind a function is that it groups part of the program and in particular, that the code within the function has some useful properties:

- a. It performs some well-defined task, which will be useful to other parts of the program.
- b. It might be useful to other programs as well; that is, we might be able to reuse it (and without having to rewrite it).
- c. The rest of the program doesn’t have to know the details of how the function is implemented. This can make the rest of the program easier to think about.
- d. By placing the code to perform the useful task into a function, and simply calling the function in the other parts of the program where the task must be performed, the rest of the program becomes clearer: rather than having some large, complicated, difficult-to-understand piece of code **repeated** wherever the task is being performed, we have a single simple function call, and the name of the function reminds us which task is being performed.
- e. Since the rest of the program doesn’t have to know the details of how the function is implemented, the rest of the program doesn’t care if the function is **re-implemented** later, in some different way (as long as it continues to perform its same task, of course!). This means that one part of the program can be rewritten, to improve performance or add a new feature (or simply to fix a bug), without having to rewrite the rest of the program.

6.1 Functions Basics

So what defines a function? It has a **name** that you call it by, and a list of zero or more **arguments** or **parameters** that you hand to it for it to act on or to direct its work; it has a **body** containing the actual

instructions (statements) for carrying out the task the function is supposed to perform; and it may give you back a **return value**, of a particular type.

Here is a very simple function, which accepts one argument, multiplies it by 2, and hands that value back:

```
int multbytwo(int x)
{
    int retval;
    retval = x * 2;
    return retval;
}
```

On the first line we see the **return type** of the function (**int**), the **name** of the function (**multbytwo**), and a list of the function's **arguments**, enclosed in parentheses. Each argument has both a name and a type; **multbytwo** accepts one argument, of type **int**, named **x**. The name **x** is arbitrary, and is used only within the definition of **multbytwo**. The caller of this function only needs to know that a single argument of type **int** is expected; the caller does not need to know what name the function will use internally to refer to that argument. In particular, the caller does not have to pass the value of a variable named **x**.

Next we see, surrounded by the familiar braces, the **body** of the function itself. This function consists of one variable declaration (**retval**) and two statements. The first statement is a conventional expression statement, which computes and assigns a value to **retval**, and the second statement is a **return** statement, which causes the function to return to its caller, and also specifies the value which the function returns to its caller.

The **return** statement can return the value of any expression, so we don't really need the **retval** variable; the function could be collapsed to:

```

int multbytwo(int x)
{
    return x * 2;
}

```

How do we call a function? We've been doing so informally since day one, but now we have a chance to call one that we've written, in full detail. Listing 6-1 introduces a tiny program to call `multbytwo` function:

Listing 6-1: Example of a User-defined Function

```

19. #include <stdio.h>

20. int multbytwo(int);

21. int main(){
22.     int i, j;
23.     i = 3;
24.     j = multbytwo(i);           //call the function
25.     printf("%d\n", j);
26.     return 0;
27. }

28. /*-----Function multbytwo-----*/
29. int multbytwo(int x){
30.     return x * 2;
31. }

```

Line 2 in the program is called a ***function prototype declaration***. It declares to the `main` function something that is defined somewhere else (here in the same source file). The function prototype declaration contains the three pieces of information about the function that a caller needs to know: the function's name, return type, and argument type(s). Since we don't care what name the `multbytwo` function will use to refer to its first argument, we don't need to mention it. On the other hand, if a function takes several arguments, giving them names in the prototype may make it easier to remember which is which, so names may optionally be used in function prototype declarations. The presence of the function prototype declaration lets the compiler know that we intend to call this function, `multbytwo`. The information in the

prototype lets the compiler generate the correct code for calling the function, and also enables the compiler to check up on our code by making sure, for example, that we pass the correct number of arguments to each function we call.

Down in the body of `main`, the action of the function call should be obvious in line 6.

```
j = multbyttwo(i);
```

calls `multbyttwo`, passing it the value of `i` as its argument. When `multbyttwo` returns, the return value is assigned to the variable `j`. Notice that the variable `i` isn't really needed, since we could just call:

```
j = multbyttwo(3);
```

And the variable `j` isn't really needed, either, since we could just call:

```
printf("%d\n", multbyttwo(3));
```

Here, the call to `multbyttwo` is a sub-expression which serves as the second argument to `printf`. The value returned by `multbyttwo` is passed immediately to `printf`.



We should say a little more about the mechanism by which an argument is passed down from a caller into a function. Formally, C is **call by value**, which means that a function receives copies of the values of its arguments. We can illustrate this with an example. Suppose, in our implementation of `multbyttwo`, we had gotten rid of the unnecessary `retval` variable like this:

```
int multbyttwo(int x){
    x = x * 2;
    return x;
}
```

We might wonder, if we wrote it this way, what would happen to the value of the variable `i` when we called

```
j = multbyttwo(i);
```

When our implementation of `multbyt看wo` changes the value of `x`, does that change the value of `i` up in the caller? ***The answer is no.*** `x` receives a copy of `i`'s value, so when we change `x` we don't change `i`.



However, there is an exception to this rule. When the argument you pass to a function is not a single variable, but is rather ***an array***, the function does not receive a copy of the array, and it therefore ***can modify the array in the caller***. The reason is that it might be too expensive to copy the entire array, and furthermore, it can be useful for the function to write into the caller's array, as a way of handing back more data than would fit in the function's single return value. We'll see an example of an array argument (which the function deliberately writes into) in the next chapter.

6.1.1 Function Prototypes

In modern C programming, it is considered good practice to use prototype declarations for all functions that you call. As we mentioned, these prototypes help to ensure that the compiler can generate correct code for calling the functions, as well as allowing the compiler to catch certain mistakes you might make.

Strictly speaking, however, prototypes are optional. You can omit the prototype but you have to define your function before your `main` function; so that the compiler will be able to recognize the function before entering the `main`. However, this is not a good practice. Usually, a good programmer keep his `main` as the first function (to be found easily) and define all other functions after it.

If prototypes are a good idea, and if we're going to get in the habit of writing function prototype declarations for functions we call that we've written (such as `multbyt看wo`), what happens for library functions such as `printf`? Where are their prototypes? The answer is in that line:

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

we've been including at the top of all of our programs. `stdio.h` is conceptually a file full of external declarations and other information pertaining to the “Standard I/O” library functions, including `printf`.

6.1.2 Function Philosophy

What makes a **good function**? The most important aspect of a good “building block” is that have a single, **well-defined task** to perform. When you find that a program is hard to manage, it's often because it has not been designed and broken up into functions cleanly. Two obvious reasons for moving code down into a function are because:

1. It **appeared** in the main program **several times**, such that by making it a function, it can be written just once, and the several places where it used to appear can be replaced with calls to the new function.
2. The main program was getting too big, so it could be made (presumably) smaller and more manageable by lopping part of it off and making it a function.

These two reasons are important, and they represent significant benefits of well-chosen functions, but they are not sufficient to automatically identify a good function. As we've been suggesting, a good function has at least these two additional attributes:

3. It does just one well-defined task, and does it well.
4. Its interface (arguments and return type) to the rest of the program is clean and narrow.



Attribute 3 is just a restatement of two things we said above. Attribute 4 says that you shouldn't have to keep track of too many things when calling a function. If you know what a function is supposed to do, and if its task is simple and well-defined, there should be just a few pieces of information you have to give it to act upon, and one or just a few pieces of information which it returns to you when it's done. If you find yourself having to pass lots and lots of information to a function, or remember details of its internal

implementation to make sure that it will work properly this time, it's often a sign that the function is not sufficiently well-defined.

The whole point of breaking a program up into functions is so that you don't have to think about the entire program at once; ideally, you can think about just one function at a time. We say that a good function is a “**black box**”. When you call a function, you only have to know what it does, not how it does it. When you're writing a function, you only have to know what it's supposed to do, and you don't have to know why or under what circumstances its caller will be calling it.

In fact, if a difficult-to-write function's **interface** is well-defined, you may be able to get away with writing a quick-and-dirty version of the function first, so that you can begin testing the rest of the program, and then go back later and rewrite the function to do the hard parts. As long as the function's original interface anticipated the hard parts, you won't have to rewrite the rest of the program when you fix the function.

6.2 Void (Non Value-Returning) Functions

Void functions are created and used just like value-returning functions except they do not return a value after the function executes. Instead of a data type, void functions use the keyword “**void**.” A void function performs a task, and then control returns back to the caller--but, it does not return a value. You may or may not use the return statement, as there is no return value. Even without the return statement, control will return to the caller automatically at the end of the function. As an example of a void function is a function to repeat the word “Hello” on the screen a certain number of times:

```
void printHello(int num){
    for(int i=0;i<num;i++)
        printf("Hello\n");
}
```

To call this function, you can simply write:

```
printHello(10);           //print Hello 10 times
```


There are a set of similarities between value-returning and void functions:

- Both: definitions can be placed before or after function `main()`...
- though, if placed after `main()` function, prototypes must be placed before `main()`
- Both: formal parameter list can be empty--though, parentheses still required
- Both: actual parameter list can use expression or variable, but must match in: type, order, number

Also there are a set of differences between value-returning and void functions:

- Void function: does not have return type
- Uses keyword **void** in function header
- Call to void function is stand-alone statement

6.3 Variables Visibility and Lifetime

We haven't said so explicitly, but variables are ***channels of communication*** within a program. You set a variable to a value at one point in a program, and at another point (or points) you read the value out again. The two points may be in adjoining statements, or they may be in widely separated parts of the program (different functions).

How long does a variable last? How widely separated can the setting and fetching parts of the program be, and how long after a variable is set does it persist? Depending on the variable and how you're using it, you might want different answers to these questions.

The visibility of a variable determines how much of the rest of the program can access that variable. You can arrange that a variable is

visible only within one part of one function, or in one function, or in one source file, or anywhere in the program. (Program code can be separated into several files)

A variable declared within the braces { } of a function is visible only within that function; variables declared within functions are called **local variables**. If another function somewhere else declares a local variable with the same name, it's a different variable entirely, and the two don't clash with each other.

On the other hand, a variable declared outside of any function is a **global variable**, and it is potentially visible anywhere within the program. You use global variables when you do want the communications path to be able to travel to any part of the program. When you declare a global variable, you will usually give it a longer, more descriptive name (not something generic like `i`) so that whenever you use it you will remember that it's the same variable everywhere.²

How long do variables last? By default, local variables (those declared within a function) have **automatic duration**: they spring into existence when the function is called, and they (and their values) disappear when the function returns. Global variables, on the other hand, have **static duration**: they last, and the values stored in them persist, for as long as the program does. (Of course, the values can in general still be overwritten, so they don't necessarily persist forever.)

Finally, it is possible to split a program up into several source files, for easier maintenance. When several source files are combined into one program, the compiler must have a way of correlating the global variables which might be used to communicate between the several source files. Furthermore, if a global variable is going to be useful for communication, there must be exactly one of it: you wouldn't want one function in one source file to store a value in one global variable named `globalvar`, and then have another function in another source file read from a different global variable named `globalvar`. Therefore,

² Another word for the visibility of variables is **variable scope**.

a global variable should have exactly one defining instance, in one place in one source file. If the same variable is to be used anywhere else (i.e. in some other source file or files), the variable is declared in those other file(s) with an **external declaration**, which is not a defining instance. The external declaration says, “hey, compiler, here's the name and type of a global variable I'm going to use, but don't define it here, don't allocate space for it; it's one that's defined somewhere else, and I'm just referring to it here.” If you accidentally have two distinct defining instances for a variable of the same name, the compiler (or the linker) will complain that it is “multi-defined”.

It is also possible to have a variable which is global in the sense that it is declared outside of any function, but private to the one source file it's defined in. Such a variable is visible to the functions in that source file but not to any functions in any other source files, even if they try to issue a matching declaration.

You get any extra control you might need over visibility and lifetime, and you distinguish between defining instances and external declarations, by using **storage classes**. A storage class is an extra keyword at the beginning of a declaration which modifies the declaration in some way. Generally, the storage class (if any) is the first word in the declaration, preceding the type name.

We said that, by default, local variables had automatic duration. To give them **static duration** (so that, instead of coming and going as the function is called, they persist for as long as the function does), you precede their declaration with the **static** keyword:

```
static int i;
```

By default, a declaration of a global variable (especially if it specifies an initial value) is the **defining instance**. To make it an external declaration, of a variable which is defined somewhere else, you precede it with the keyword **extern**:

```
extern int j;
```

Finally, to arrange that a global variable is visible only within its containing source file, you precede it with the static keyword:



```
static int k;
```

Notice that the static keyword can do two different things: it adjusts the duration of a local variable from automatic to static, or it adjusts the visibility of a global variable from truly global to private-to-the-file.

To summarize, we've talked about two different attributes of a variable: visibility and duration. These are orthogonal, as shown in Table 6-1.


Table 6-1: Variables Visibility and Lifetime

	Duration:	
Visibility:	Automatic	Static
Local	normal local variables	static local variables
Global	N/A	normal global variables

-  We can distinguish between file-scope global variables and truly global variables, based on the presence or absence of the static keyword.
-  We can distinguish between external declarations and defining instances of global variables, based on the presence or absence of the extern keyword.

6.3.1 Default Initialization

The duration of a variable (whether static or automatic) also affects its default initialization.

 If you do not explicitly initialize them, automatic-duration variables (that is, local, non-static ones) are ***not guaranteed to have any particular initial value***; they will typically contain garbage. It is therefore a fairly serious error to attempt to use the value of an automatic variable which has never been initialized or assigned to: the program will either work incorrectly, or the garbage value may just happen to be “correct” such that the program appears to work correctly! However, the particular value that the garbage takes on can vary depending literally on anything: other parts of the program, which compiler was used, which hardware or operating system the program is running on. So you hardly want to say that a program which uses an uninitialized variable “works”; it may seem to work, but it works for the wrong reason, and it may stop working tomorrow.

Static-duration variables (global and static local), on the other hand, are guaranteed to be initialized to ***0 (zero)*** if you do not use an explicit initializer in the definition.

6.3.2 Examples

Listing 6-2 provides an example demonstrating almost everything we've seen so far in this section. Here we have six variables, three declared outside and three declared inside of the function `f()`.

`globalvar` is a global variable. The declaration we see is its defining instance that includes an initial value (What about if we remove this initialization?). `globalvar` can be used anywhere in this source file, and it could be used in other source files, too (as long as corresponding external declarations are issued in those other source files).

`anotherglobalvar` is a second global variable. It is not defined here; the defining instance for it (and its initialization) is somewhere else (another source file).

Listing 6-2: Example of Variables Visibility

```
1. int globalvar = 1;
2. extern int anotherglobalvar;
3. static int privatevar;
4. f(){
5.         int localvar;
6.         int localvar2 = 2;
7.         static int persistentvar;
8. }
```

privatevar is a “private” global variable. It can be used anywhere within this source file, but functions in other source files cannot access it, even if they try to issue external declarations for it. Since it has static duration and receives ***no explicit initialization***, **privatevar** will be initialized to 0.

localvar is a local variable within the function **f()**. It can be accessed only within the function **f()**. (If any other part of the program declares a variable named “**localvar**”, that variable will be distinct from the one we’re looking at here). **localvar** is conceptually “created” each time **f()** is called, and disappears when **f()** returns. Any value which was stored in **localvar** last time **f()** was running will be lost and will not be available next time **f()** is called. Furthermore, since it has no explicit initializer, the value of **localvar** will in general be garbage each time **f()** is called.

localvar2 is also local, and everything that we said about **localvar** applies to it, except that since its declaration includes an explicit initializer, it will be initialized to 2 each time **f()** is called.

Finally, **persistentvar** is again local to **f()**, but it does maintain its value between calls to **f()**. It has ***static duration*** but no explicit initializer, so its initial value will be 0.

Don't worry about static variables for now if they don't make sense to you; they're a relatively sophisticated concept, which you won't need to use at first.

6.4 Exercises

- 1) What are the four important parts of a function? Which three does a caller need to know?
- 2) What is the difference between a defining instance and an external declaration?
- 3) What is the difference between static and automatic duration?
- 4) Given the following code fragment, what is the value of x after the function count() is called and executed twice?

```
int count(){
    static int x = 0;
    x += 1;
}
```

- a) 0
 - b) 1
 - c) 2
 - d) more than 2
- 5) What is the purpose of the term 'return type' in a function definition?
- a) To specify the data type of the function.
 - b) To specify the type of data the function returns.
 - c) to specify the type of variables used as parameters.
 - d) To specify the type of variables passed to the function.

- 6) Which one of the following best describes a function call?
- a) Identifies the type of data returned from a function and the function's storage class.
 - b) Defines the operation a function must perform.
 - c) Identifies the function parameters.
 - d) States the name of the function and passes arguments.
- 7) Which one of the following best describes a function definition?
- a) Allocates a storage location for the return value.
 - b) Contains the statements that indicate the function name and parameters to be used in processing.
 - c) Calls a predefined function and passes the necessary arguments.
 - d) Tells the compiler which function you are using.
- 8) The arguments passed to a function are
- a) global variables visible to all functions
 - b) local variables to the receiving function
 - c) not allowed to be used by the receiving function
 - d) available only in ANSI C
- 9) Write code to sum the elements of an array of int. (Write it as a function) Use it to sum the array (write a main for the program):
- ```
int a[] = {1, 2, 3, 4, 5, 6};
```



- 10) Write a loop to call the `multbytwo()` function (Listing 6-1: Example of a User-defined Function) on the numbers 1-10.
- 11) Write a `square()` function and use it to print the squares of the numbers 1-10:
 

```
1 1
2 4
3 9
4 16
...
9 81
10 100
```
- 12) Write the function
 

```
void printnchars(int ch, int n)
```

 which is supposed to print the character `ch`, `n` times. (Remember that `%c` is the `printf` format to use for printing characters.) For example, the call `printnchars('x', 5)` would print 5 `x`'s. Use this function to rewrite the triangle-printing program of assignment 26 (exercises of Chapter 4).
- 13) Write a function to compute the factorial of a number, and use it to print the factorials of the numbers 1-7.
- 14) Write a function `celsius()` to convert degrees Fahrenheit to degrees Celsius. (The conversion formula is  $^{\circ}\text{C} = 5/9 * (^{\circ}\text{F} - 32)$ .) Use it to print a Fahrenheit-to-Centigrade table for -40 to 220 degrees Fahrenheit, in increments of 10 degrees. (Remember that `%f` is the `printf` format to use for printing floating-point numbers. Also, remember that the integer expression `5/9` gives 0, so you won't want to use integer division.)
- 15) Write two functions
 

```
randrange(int n)
```

 which returns random integers from 1 to `n`, or the function
 

```
int randrange2(int m, int n)
```

 which returns random integers in the range `m` to `n`.  
  
 The header file `<stdlib.h>` defines a constant, `RAND_MAX`, which is the maximum number returned by the `rand()` function. A better way of reducing the range of the `rand()` function is like this:
 

```
rand() / (RAND_MAX / N + 1)
```

 (where `N` is the range of numbers you want).



## 7 Basic Input and Output

So far, we've been using `printf` to do output, and we haven't had a way of doing any input. In this chapter, we'll learn a bit more about `printf`, and we'll begin learning about character-based input and output, reading command lines and strings.

### 7.1 “printf” Function

`printf`'s name comes from *print formatted*. It generates output under the control of a *format string* (its first argument) which consists of literal characters to be printed and also special character sequences (format specifiers) that request that other arguments be fetched, formatted, and inserted into the string. Our very first program was nothing more than a call to `printf`, printing a constant string:

```
printf("Hello, world!\n");
```

Our second program also featured a call to `printf`:

```
printf("i is %d\n", i);
```

In that case, whenever `printf` prints the string `"i is %d"`, it did not print it as it is; it replaces the two special characters `%d` with the value of the variable `i`. There are quite a number of format specifiers for `printf`. Table 7-1 lists the basic ones.

It is also possible to specify the width and precision of numbers and strings as they are inserted. For example, a notation like `%3d` means to print an `int` in a field at least 3 spaces wide; a notation like `%5.2f` means to print a float or double in a field at least 5 spaces wide, with two places to the right of the decimal.

**Table 7-1: Format Specifiers**

| Format Specifier | Function                                       |
|------------------|------------------------------------------------|
| %d               | print an int argument in decimal               |
| %ld              | print a long int argument in decimal           |
| %c               | print a character                              |
| %s               | print a string                                 |
| %f               | print a float or double argument               |
| %e               | same as %f, but use exponential notation       |
| %o               | print an int argument in octal (base 8)        |
| %x               | print an int argument in hexadecimal (base 16) |
| %%               | print a single %                               |

To illustrate with a few more examples, the call

```
printf("%c %2d %f %e %s %d%%\n", '1', 2, 3.14, 56000000.,
 "eight", 9);
```

would print

```
1 02 3.140000 5.600000e+07 eight 9%
```

The call

```
printf("%d %o %x\n", 100, 100, 100);
```

would print

```
100 144 64
```

Successive calls to **printf** just build up the output a piece at a time, so the calls

```
printf("Hello, ");

printf("world!\n");
```

would also print Hello, world! (on one line of output).

Earlier we learned that C represents characters internally as small integers corresponding to the characters' values in the machine's character set (typically ASCII). This means that there isn't really much difference between a character and an integer in C; most of the difference is in whether we choose to interpret an integer as an integer or a character. **printf** is one place where we get to make that choice: **%d** prints an integer value as a string of digits representing its decimal value, while **%c** prints the character corresponding to a character set value. So the lines

```
char c = 'A';

int i = 97;

printf("c = %c, i = %d\n", c, i);
```

would print **c** as the character **A** and **i** as the number **97**. But if, on the other hand, we called

```
printf("c = %d, i = %c\n", c, i);
```

we'd see the decimal value 65 (printed by **%d**) of the character **'A'**, followed by the character (**'a'**) that happens to have the decimal value **97**.



You have to be careful when calling **printf**. It has no way of knowing how many arguments you've passed it or what their types are other than by looking for the format specifiers in the format string. If there are more format specifiers (that is, more **%** signs) than there are arguments, or if the arguments have the wrong types for the format specifiers, **printf** can misbehave badly, often printing nonsense numbers or (even worse) numbers which mislead you into thinking that some other part of your program is broken.

## 7.2 Character Input and Output

Unless a program can read some input, it's hard to keep it from doing exactly the same thing every time it's run, and thus being rather boring after a while. The most basic way of reading input is by calling the function `getchar`. `getchar` reads one character from the “*standard input*” which is usually the user's keyboard, but which can sometimes be redirected by the operating system. `getchar` returns (rather obviously) the character it reads, or, if there are no more characters available, the special value `EOF` (*end of file*).

A companion function is `putchar` that writes one character to the “*standard output*” (usually the user's screen). Using these two functions, we can write a very basic program as the one shown in Listing 7-1 to copy the input, a character at a time, to the output:

**Listing 7-1: A Program to Copy Input to Output**

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

2. /* copy input to output */

3. int main(){
4. int c;

5. c = getchar();

6. while(c != EOF){
7. putchar(c);
8. c = getchar();
9. }

10. return 0;
11. }
```

This code is straightforward, and I encourage you to type it in and try it out. It reads one character, and if it is not the `EOF` code, enters a `while` loop, printing one character and reading another, as long as the character read is not `EOF`. This is a straightforward loop, although

there's one mystery surrounding the declaration of the variable `c`: if it holds characters, why is it an `int`?

We said that a `char` variable could hold integers corresponding to character set values, and that an `int` could hold integers of more arbitrary values (up to  $\pm 32767$ ). Since most character sets contain a few hundred characters, an `int` variable can in general comfortably hold all `char` values. Therefore, there's nothing wrong with declaring `c` as an `int`. But in fact, it's important to do so, because `getchar` can return every character value, plus that special, non-character value `EOF`, indicating that there are no more characters. Type `char` is only guaranteed to be able to hold all the character values; it is not guaranteed to be able to hold this “no more characters” value without possibly mixing it up with some actual character value. Therefore, you should always remember to use an `int` for anything you assign `getchar`'s return value to.

When you run the character copying program, and it begins copying its input (your typing) to its output (your screen), you may find yourself wondering ***how to stop it***. It stops when it receives end-of-file (`EOF`), but how do you send `EOF`? The answer depends on what kind of computer you're using. On Linux systems, it's almost always [control-D]. On MS-DOS machines, it's [control-Z] followed by the [RETURN] key.

Finally, don't be disappointed the first time you run the character copying program. You'll type a character, and see it on the screen right away, and assume it's your program working, but it's only your computer echoing every key you type, as it always does. When you hit [RETURN], a full line of characters is made available to your program. It then zips several times through its loop, reading and printing all the characters in the line in quick succession.

In other words, when you run this program, it will probably seem to copy the input a line at a time, rather than a character at a time. You may wonder how a program could instead read a character right away, without waiting for the user to hit [RETURN]. That's an excellent question, but unfortunately the answer is beyond the scope of our

discussion here (but for sure you are free to search the Internet for a solution).

Stylistically, the character-copying program above can be said to have one minor flaw: it contains two calls to **getchar**, one which reads the first character and one which reads (by virtue of the fact that it's in the body of the loop) all the other characters. This seems inelegant and perhaps unnecessary, and it can also be risky: if there were more things going on within the loop, and if we ever changed the way we read characters, it would be easy to change one of the **getchar** calls but forget to change the other one. Is there a way to rewrite the loop so that there is only one call to **getchar**, responsible for reading all the characters? Is there a way to read a character, test it for **EOF**, and assign it to the variable **c**, all at the same time?

Yes, there is. It relies on the fact that the assignment operator, **=**, is just another operator in C. An assignment is not (necessarily) a standalone statement; it is an expression, and it has a value (the value that's assigned to the variable on the left-hand side), and it can therefore participate in a larger, surrounding expression. Therefore, most C programmers would write the character-copying loop like this:

```
while((c = getchar()) != EOF)
 putchar(c);
```

What does this mean? The function **getchar** is called, as before, and its return value is assigned to the variable **c**. Then the value is immediately compared against the value **EOF**. Finally, the true/false value of the comparison controls the while loop: as long as the value is not **EOF**, the loop continues executing, but as soon as an **EOF** is received, no more trips through the loop are taken, and it exits. The net result is that the call to **getchar** happens inside the test at the top of the while loop, and doesn't have to be repeated before the loop or within the loop. The extra parentheses around

```
(c = getchar())
```



are important, and are there because the precedence of the **!=** operator is higher than that of the **=** operator. If we wrote:

```
while(c = getchar() != EOF) /* WRONG */
```

the compiler would interpret it as

```
while(c = (getchar() != EOF))
```

That is, it would assign the result of the `!=` operator to the variable `c`, which is not what we want.

## 7.2.1 Reading Lines

It's often convenient for a program to process its input not a character at a time but rather a line at a time, that is, to read an entire line of input and then act on it all at once. We're going to learn more about character input and about writing functions in general by writing our own function to read one line as shown in Listing 7-2.

### Listing 7-2: Function to Read One Line

```
1. #include <stdio.h>
2. /* Read one line from standard input, */
3. /* copying it to an array (no more than max chars). */
4. /* Returns line length, or 0 for empty line, or EOF for
 end-of-file. */

5. int getline(char line[], int max){
6. int nch = 0, c=0;
7. max = max - 1; /* leave room for '\0' */

8. while((c = getchar()) != EOF){
9. if(c == '\n')
10. break;

11. if(nch < max){
12. line[nch] = c;
13. nch = nch + 1;
14. }
15. }
16. if(c == EOF && nch == 0)
17. return EOF;
18. line[nch] = '\0';
19. return nch;
20. }
```



As the comments in lines from 2 to 4 indicate, this function will read one line of input from the standard input, placing it into the `line` array. The size of the line array is given by the `max` argument; the function will never write more than `max` characters into `line`.

The main body of the function is a `getchar` loop, much as we used in the character-copying program. In the body of this loop, however, we're storing the characters in an array (rather than immediately printing them out). Also, we're only reading one line of characters, then stopping and returning. There are several new things to notice here.



First of all, the `getline` function accepts an **array as a parameter**. As we've said, array parameters are an exception to the rule that functions receive copies of their arguments--in the case of arrays, the function does have access to the actual array passed by the caller, and can modify it. Since the function is accessing the caller's array, not creating a new one to hold a copy, the function does not have to declare the argument array's size; it's set by the caller (Thus, the brackets in `char line[]` are empty). However, so that we won't overflow the caller's array by reading too long a line into it, we allow the caller to pass along the size of the array that we promise not to exceed.

Second, we see an example of the `break` statement. The top of the loop looks like our earlier character-copying loop--it stops when it reaches `EOF`--but we only want this loop to read one line, so we also stop (that is, break out of the loop) when we see the `\n` character signifying end-of-line.

We haven't learned about the internal representation of strings yet, but it turns out that **strings in C are simply arrays of characters**, which is why we are reading the line into an array of characters. The end of a string is marked by the special character, `'\0'`. To make sure that there's always room for that character, on our way in we subtract 1 from `max`, the argument that tells us how many characters we may place in the line array. When we're done reading

the line, we store the end-of-string character `'\0'` at the end of the string we've just built in the line array.

Finally, there's one part in the code which isn't too important for our purposes now but which you may wonder about: it's arranged to handle the possibility that a few characters (i.e. the apparent beginning of a line) are read, followed immediately by an `EOF`, without the usual `\n` end-of-line character. (That's why we return `EOF` only if we received `EOF` and we hadn't read any characters first.)

In any case, the function returns the length (number of characters) of the line it read, not including the `\n`. (Therefore, it returns 0 for an empty line.) Like `getchar`, it returns `EOF` when there are no more lines to read. (It happens that `EOF` is a negative number, so it will never match the length of a line that `getline` has read.)

Here is an example of a test program which calls `getline`, reading the input a line at a time and then printing each line back out:

```
#include <stdio.h>

extern int getline(char [], int);

main()
{
 char line[256];

 while(getline(line, 256) != EOF)
 printf("you typed \"%s\\n\"", line);

 return 0;
}
```

The notation `char []` in the function prototype for `getline` says that `getline` accepts as its first argument an array of `char`. When the program calls `getline`, it is careful to pass along the actual size of the array. You might notice a potential problem: since the number 256 appears in two places, if we ever decide that 256 is too small, and that we want to be able to read longer lines, we could easily change one of the instances of 256, and forget to change the other one. (Can you think about a solution to avoid this sort of problem?)

## 7.2.2 Reading Numbers

The `getline` function of the previous subsection reads one line from the user, as a string. What if we want to read a number? One straightforward way is to read a string as before, and then immediately convert the string to a number. The standard C library contains a number of functions for doing this. The simplest to use are `atoi()`, which converts a string to an integer, and `atof()`, which converts a string to a floating-point number. (Both of these functions are declared in the header `<stdlib.h>`, so you should `#include` that header at the top of any file using these functions.) You could read an integer from the user like this:

```
#include <stdlib.h>

char line[256];
int n;
printf("Type an integer:\n");
getline(line, 256);
n = atoi(line);
```

Now the variable `n` contains the number typed by the user. (This assumes that the user did type a valid number, and that `getline` did not return `EOF`). Reading a floating-point number is similar:

```
#include <stdlib.h>

char line[256];
double x;
printf("Type a floating-point number:\n");
getline(line, 256);
x = atof(line);
```

Another way of reading in numbers, which you're likely to see in other books on C, involves the `scanf` function, but it has several problems, so we won't discuss it here.

## 7.3 Strings

Strings in C are represented by *arrays of characters*. The end of the string is marked with a special character, the **null** character, which is simply the character with the value 0. The null or string-terminating character is represented by another character escape sequence, `\0`. (We've seen it once already, in the `getline` function). C has few built-in facilities for manipulating strings.



Notice that whenever we write a string, enclosed in double quotes, C automatically creates an array of characters for us, containing that string, terminated by the `\0` character. For example, we can declare and define an array of characters, and initialize it with a string constant:

```
char string[] = "Hello, world!";
```

In this case, we can leave out the dimension of the array, since the compiler can compute it for us based on the size of the initializer (14, including the terminating `\0`). This is the only case where the compiler sizes a string array for us, however; in other cases, it will be necessary that we decide how big the arrays and other data structures we use to hold strings are. To do anything else with strings, we must typically call functions. The C library contains a few basic string manipulation functions, and to learn more about strings, we'll be looking at how these functions might be implemented.

Since C never lets us assign entire arrays, we use the `strcpy` function to copy one string to another:

```
#include <string.h>
```

```
char string1[] = "Hello, world!";
char string2[20];
```

```
strcpy(string2, string1);
```

The *destination* string is `strcpy`'s first argument, so that a call to `strcpy` mimics an assignment expression (with the destination on the left-hand side). Notice that we had to allocate `string2` big enough to hold the string that would be copied to it. Also, at the top of any source

file where we're using the standard library's string-handling functions (such as **strcpy**) we must include the line

```
#include <string.h>
```

that contains external declarations for these functions.

Since C won't let us compare entire arrays, either, we must call a function to do that, too. The standard library's **strcmp** function compares two strings, and returns 0 if they are identical, or a negative number if the first string is alphabetically “less than” the second string, or a positive number if the first string is “greater.” (Roughly speaking, what it means for one string to be “less than” another is that it would come first in a dictionary or a telephone book). Here is an example:

```
char string3[] = "this is";
char string4[] = "a test";

if(strcmp(string3, string4) == 0)
 printf("strings are equal\n");
else
 printf("strings are different\n");
```

This code fragment will print “strings are different”.

Another standard library function is **strcat** that concatenates strings. It does not concatenate two strings together and give you a third, new string; what it really does is append one string onto the end of another. (If it gave you a new string, it would have to allocate memory for it somewhere, and the standard library string functions generally never do that for you automatically). Here's an example:

```
char string5[20] = "Hello, ";
char string6[] = "world!";

printf("%s\n", string5);

strcat(string5, string6);

printf("%s\n", string5);
```

The first call to `printf` prints “Hello, “, and the second one prints “Hello, world!”, indicating that the contents of `string6` have been tacked on to the end of `string5`. Notice that we declared `string5` with extra space, to make room for the appended characters.

If you have a string and you want to know its length, you can call `strlen` that returns the length of the string (i.e. the number of characters in it), not including the `\0`:

```
char string7[] = "abc";

int len = strlen(string7);

printf("%d\n", len);
```

Finally, you can print strings out with `printf` using the `%s` format specifier, as we’ve been doing in these examples already (e.g. `printf("%s\n", string5);`).

Since a string is just an array of characters, all of the string-handling functions we’ve just seen can be written quite simply, using no techniques more complicated than the ones we already know. In fact, it’s quite instructive to look at how these functions might be implemented. Here is a version of `strcpy`:

```
mystrcpy(char dest[], char src[]){
 int i = 0;

 while(src[i] != '\0'){
 dest[i] = src[i];
 i++;
 }

 dest[i] = '\0';
}
```

We’ve called it `mystrcpy` instead of `strcpy` so that it won’t clash with the version that’s already in the standard library. Its operation is simple: it looks at characters in the `src` string one at a time, and as long as they’re not `\0`, assigns them, one by one, to the corresponding positions in the `dest` string. When it’s done, it terminates the `dest` string by appending a `\0`. (After exiting the while loop, `i` is

guaranteed to have a value one greater than the subscript of the last character in **src**).

Here is a version of **strcmp**:

```
mystrcmp(char str1[], char str2[]){
 int i = 0;

 while(1){
 if(str1[i] != str2[i])
 return str1[i] - str2[i];
 if(str1[i] == '\0' && str2[i] == '\0')
 return 0;
 i++;
 }
}
```

Characters are compared one at a time. If two characters in one position differ, the strings are different, and we are supposed to return a value less than zero if the first string (**str1**) is alphabetically less than the second string. Since characters in C are represented by their numeric character set values, and since most reasonable character sets assign values to characters in alphabetical order, we can simply subtract the two differing characters from each other: the expression **str1[i] - str2[i]** will yield a negative result if the *i*'th character of **str1** is less than the corresponding character in **str2**. (As it turns out, this will behave a bit strangely when comparing upper- and lower-case letters, but it's the traditional approach, which the standard versions of **strcmp** tend to use.) If the characters are the same, we continue around the loop, unless the characters we just compared were (both) **\0**, in which case we've reached the end of both strings, and they were both equal. Notice that we used what may at first appear to be an infinite loop--the controlling expression is the constant **1**, which is always true. What actually happens is that the loop runs until one of the two return statements breaks out of it (and the entire function). Note also that when one string is longer than the other, the first test will notice this (because one string will contain a real character at the [*i*] location, while the other will contain **\0**, and these are not equal) and the return value will be computed by subtracting the real

character's value from 0, or vice versa. (Thus the shorter string will be treated as “less than” the longer.)

Finally, here is a version of `strlen`:

```
int mystrlen(char str[]){
 int i;

 for(i = 0; str[i] != '\0'; i++)
 {}

 return i;
}
```

In this case, all we have to do is find the `\0` that terminates the string, and it turns out that the three control expressions of the for loop do all the work; there's nothing left to do in the body. Therefore, we use an empty pair of braces `{}` as the loop body. Equivalently, we could use a null statement, which is simply a semicolon:

```
for(i = 0; str[i] != '\0'; i++) ;
```

Notice that there is a big difference between a character and a string, even a string which contains only one character (other than the `\0`). For example, `'A'` is not the same as `"A"`. To drive home this point, let's illustrate it with a few examples. If you have a string:

```
char string[] = "hello, world!";
```

you can modify its first character by saying

```
string[0] = 'H';
```

Since you're replacing a character, you want a character constant, `'H'`. It would not be right to write

```
string[0] = "H"; /* WRONG */
```

because `"H"` is a string (an array of characters), not a single character. (The destination of the assignment, `string[0]`, is a `char`, but the right-hand side is a string; these types don't match.)

On the other hand, when you need a string, you must use a string. To print a single newline, you could call



```
printf("\n");
```

It would not be correct to call

```
printf('\n'); /* WRONG */
```

`printf` always wants a string as its first argument. As one final example, `putchar` wants a single character, so `putchar('\n')` would be correct, and `putchar("\n")` would be incorrect.

We must also remember the difference between strings and integers. If we treat the character '1' as an integer, perhaps by saying:

```
int i = '1';
```

we will probably not get the value 1 in `i`; we'll get the value of the character '1' in the machine's character set. (In ASCII, it's 49) When we do need to find the numeric value of a digit character (or to go the other way, to get the digit character with a particular value) we can make use of the fact that, in any character set used by C, the values for the digit characters, whatever they are, are contiguous. In other words, no matter what values 'o' and '1' have, '1' - 'o' will be 1 (and, obviously, 'o' - 'o' will be 0). So, for a variable `c` holding some digit character, the expression

```
c - '0'
```

gives us its value. (Similarly, for an integer value `i`, `i + '0'` gives us the corresponding digit character, as long as `0 <= i <= 9`)

Just as the character '1' is not the integer 1, the string "123" is not the integer 123. When we have a string of digits, we can convert it to the corresponding integer by calling the standard function `atoi`:

```
char string[] = "123";
int i = atoi(string);
int j = atoi("456");
```

## 7.4 Exercises

1) What output does each of these produce?

- a) `putchar('a');`
- b) `putchar('\007');`
- c) `putchar('\n');`
- d) `putchar('\t');`
- e) `n = 32; putchar(n);`
- f) `putchar('\');`

2) For the different values of `n`, what is the output?

```
printf("%x %c %o %d",n,n,n,n);
```

- a) `n = 67`
- b) `n = 20`
- c) `n = 128`
- d) `n = 255`
- e) `n = 100`

3) What is wrong with each of these?

- a) `#include stdio.h`
- b) `putchar('/n');`
- c) `printf("\nPhone Number: (%s) %s",phone_number);`
- d) `getch(ch);`
- e) `putch() = ch;`

- 4) Given the following character array, what does `state[4]` reference?

```
char state[5][3] = {"AA", "BB", "CC", "DD", "EE"};
```

- a) The address of the first character in the string CC.
  - b) The address of the first character in the string DD.
  - c) The address of the first character in the string EE.
  - d) None of the above,
- 5) Given the following character array, what does the `state[3][1]` reference?

```
char state[5][3] = {"AT", "BU", "CV", "DX", "EY"};
```

- a) The letter X.
  - b) The letter Y.
  - c) The letter V.
  - d) The letter D.
- 6) If Charles is compared to Charley, using the `strcmp()` function, the value returned is
- a) 0
  - b) > 0
  - c) < 0
  - d) -1
- 7) What is the purpose of the `strcpy()` function?
- a) To assign the value of string2 to string1.
  - b) To determine if string2 is larger than string1.
  - c) To assign the value of string1 to string2.
  - d) To determine if string1 is larger than string2.

- 8) How many bytes of memory would the compiler allocate for the following statement?

```
char names[][5] = {"Ali ", "Amal", "Mai "};
```

- a) 15
  - b) 24
  - c) 4
  - d) 10
- 9) What would the expression  
    c = getchar() != EOF  
do?
- 10) Why must the variable used to hold getchar's return value be type int?
- 11) Write a program which reads lines (using getline), converts each line to an integer using atoi, and computes the average of all the numbers read. (Like the example programs in the chapter, it should determine the end of by checking for EOF). Remember that integer division truncates, so you'll have to declare some of your variables as float or double.
- 12) Write a rudimentary checkbook balancing program. It will use getline to read a line, which will contain either the word "check" or "deposit". The next line will contain the amount of the check or deposit. After reading each pair of lines, the program should compute and print the new balance. You can declare the variable to hold the running balance to be type float, and you can use the function atof (also in the standard library) to convert an amount string read by getline into a floating-point number. When the program reaches end-of-file while reading "check" or "deposit", it should exit. (In outline, the program will be somewhat similar to the average-finding program.)

For example, given the input

```
deposit
100
check
```

```
12.34
check
49.00
deposit
7.01
```

the program should print something like

```
balance: 100.00
balance: 87.66
balance: 38.66
balance: 45.67
```

- 13) Write a program to read its input, one character at a time, and print each character and its decimal value.
- 14) Write a program to read its input, one line at a time, and print each line backwards. To do the reversing, write a function
- ```
void reverse(char line[], int len)
{
    ...
}
```



8 User Defined Types: Structures

So far, we have been using C's basic types (char, int, long int, double, etc.) and a few derived types (arrays of basic types, and functions returning basic types). In this chapter, we'll learn about another way to derive more complex types: by building user-defined types (structures).

User-defined data types have a little bit restriction: you don't have ultimate flexibility; you can define your own data types any way you want, as long as they're collections of other types. What you couldn't define would be data types that held, say, tractors or teddy bears, because of course computers have no way of holding those objects. You're ultimately restricted to the primitive types of data that computers can represent, and C's basic types cover most of those.

Very roughly speaking, a structure is a little bit like an array. An array is a collection of associated values, all of the same type. A structure is a collection of associated values, but the values can all have *different types*.

8.1 Structures

The basic user-defined data type in C is the **structure**, or **struct**. Defining structures is a two-step process: first you define a “**template**” which describes the new type, and then you declare variables having the new type (or functions returning the new type, etc.).

As a simple example, suppose we wanted to define our own type for representing complex numbers. A complex number consists of a real and imaginary part, where the imaginary part is some multiple of the square root of negative 1. (You don't have to understand complex numbers to understand this example; you can think of the real and imaginary parts as the x and y coordinates of a point on a plane). Since

a complex number consists of a real and imaginary part, we need a way of holding both these quantities in one data type, and a structure will do just the trick. Here is how we might declare our complex type:

```
struct complex{
    double real;
    double imag;
};
```

A structure declaration consists of up to four parts, of which we can see three in the example above. The first part is the keyword **struct** which indicates that we are talking about a structure. The second part is a **name** or tag by which this structure (that is, this new data type) will be known. The third part is a list of the structure's **members** (also called components or fields). This list is enclosed in braces {}, and contains what look like the declarations of ordinary variables. Each member has a name and a type, just like ordinary variables, but here we are not declaring variables; we are setting up the structure of the structure by defining the collection of data types which will make up the structure. Here we see that the complex structure will be made up of two members, both of type double, one named **real** and one named **imag**.

It's important to understand that what we've defined here is just the new data type; we have not yet declared any variables of this new type! The name **complex** (the second part of the structure declaration) is not the name of a variable; it's the name of the structure type. The names **real** and **imag** are not the names of variables; they're identifiers for the two components of the structure. A structure template (in this case **complex**) does not occupy any memory space and does not have an address, it is simply a description of a new data type.

Storage is allocated for the structure when a variable of that structure type is declared. We declare variables of our new complex type with declarations like these:

```
struct complex c1;
```

or

```
struct complex c2, c3;
```

These look almost like our previous declarations of variables having basic types, except that instead of a type keyword like **int** or **double**, we have the two-word type name **struct complex**. The keyword **struct** indicates that we're talking about a structure, and the identifier **complex** is the name for the particular structure we're talking about. **c1**, **c2**, and **c3** will all be declared as variables of type **struct complex**; each one of them will have real and imaginary parts buried inside them.

Notice that when we define structures in this way we have not quite defined a new type on a par with **int** or **double**. We cannot say:

```
complex c1;           /* WRONG */
```

The name **complex** does not become a full-fledged type name like **int** or **double**; it's just the name of a particular structure, and we must use the keyword **struct** and the name of a particular structure (e.g. **complex**) to talk about that structure type. (in C++ a new structure does automatically become a full-fledged type)

I said that a structure definition consisted of up to four parts. We saw the first three of them in the first example; the fourth part of a full structure declaration is simply a list of variables, which are to be declared as having the structure type at the same time as the structure itself is defined. For example, if we had written

```
struct complex{  
    double real;  
    double imag;  
} c1, c2, c3;
```

we would have defined the type **struct complex**, and right away declared three variables **c1**, **c2**, and **c3** all of type **struct complex**.



Because a structure definition can also declare variables, it's important not to forget the semicolon at the end of a structure definition. If you accidentally write:


```
struct complex{
    double real;
    double imag;
}
```

without a semicolon, the compiler will keep looking for something later in the file and try to declare it as being of type **struct complex**, which will either result in a confusing error message or (if the compiler succeeds) a confusing miss-declaration.

8.2 Accessing Members of Structures

We said that a structure was a little bit like an array: a collection of members (elements). We access the elements of an array by using a numeric index in square brackets []. We access the elements of a structure by name, using the **structure selection operator** which is a dot (a period). The structure selection operator is a little like the other binary operators we've seen, but much more restricted: on its left must be a variable or object of structure type, and on its right must be the name of one of the members of that structure. For example, if **c1** is a variable of type **struct complex** as declared in the previous section, then **c1.real** is its real part and **c1.imag** is its imaginary part.

Like indexed array references, references to the members of structure variables (using the structure selection operator) can appear anywhere, either on the right or left side of assignment operators. We could say:

```
c1.real = 1
```

to set the real part of **c1** (that is, the real member within **c1**) to 1, or:

```
c1.imag = c2.imag
```

to fetch the imaginary part of **c2** and assign it to the imaginary part of **c1**, or:

```
c1.real = c2.real + c3.real
```

to take the real parts of **c2** and **c3**, add them together, and assign the result to the real part of **c1**.

8.3 Operations on Structures

There are a relatively small number of operations which C directly supports on structures. As we've seen, we can define structures, declare variables of structure type, and select the members of structures. We can also assign entire structures: the expression

```
c1 = c2
```

would assign all of **c2** to **c1** (both the real and imaginary parts, assuming the preceding declarations). We can also pass structures as arguments to functions, and declare and define functions which return structures. But to do anything else, we typically have to write our own code (often as functions). For example, we could write a function to add two complex numbers:

```
struct complex
cpx_add(struct complex c1, struct complex c2)
{
    struct complex sum;
    sum.real = c1.real + c2.real;
    sum.imag = c1.imag + c2.imag;
    return sum;
}
```

We could then say things like

```
c1 = cpx_add(c2, c3)
```

One more thing you can do with a structure is initialize a structure variable while declaring it. *As for array initializations*, the initializer consists of a comma-separated list of values enclosed in braces {}:

```
struct complex c1 = {1, 2};  
struct complex c2 = {3, 4};
```

Of course, the type of each initializer in the list must be compatible with the type of the corresponding structure member.

8.3.1 Nested Structures

A structure variable can be a member of another structure template.

```
struct vital{  
    int age;  
    int height;  
};  
struct home{  
    char name[12];  
    char address[20];  
};  
struct person{  
    struct vital emp;  
    struct home place;  
} employ;
```

In order to access a member of one of the nested structures, the dot operator is used until the lowest member is reached in the structure hierarchy:

```
printf("My name is: %s ", employ.place.name);  
printf("My age is: %d ", employ.emp.age);
```

8.3.2 Arrays of Structures

Like any other data type in C, a variable of a structure type can be arrayed.

```
struct person
{
    struct vital emp;
    struct home place;
} stats[100];
```

The above declaration would allow for the storage of 100 items of type people. Any specific item could be referenced as:

```
stats[indx].place.name
```

```
stats[indx].emp.age
```

Notice stats is the arrayed item, and requires the array operator, [].

8.4 Define a Type New Name: typedef

The C programming language provides a keyword called **typedef**, which you can use to give a type a new name. Following is an example to define a term BYTE for one-byte numbers:

```
typedef unsigned char BYTE;
```

After this type definitions, the identifier **BYTE** can be used as an abbreviation for the type unsigned char, for example:

```
BYTE b1, b2;
```

By convention, uppercase letters are used for these definitions to remind the user that the type name is really a symbolic abbreviation, but you can use lowercase, as follows:

```
typedef unsigned char byte;
```

You can use **typedef** to give a name to user defined data type as well. For example you can use **typedef** with structure to define a new data type and then use that data type to define structure variables directly as follows:

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

typedef struct Books{
    char   title[50];
    char   author[50];
    char   subject[100];
    int    book_id;
} Book;

int main( )
{
    Book book;

    strcpy( book.title, "C Programming");
    strcpy( book.author, "Mahmoud El-Gayyar");
    strcpy( book.subject, "C Programming Tutorial");
    book.book_id = 6495407;

    printf( "Book title : %s\n", book.title);
    printf( "Book author : %s\n", book.author);
    printf( "Book subject : %s\n", book.subject);
    printf( "Book book_id : %d\n", book.book_id);

    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Book title : C Programming
Book author : Mahmoud El-Gayyar
Book subject : C Programming Tutorial
Book book_id : 6495407
```

8.5 Exercises

- 1) What are the four parts of a structure definition?
- 2) Write a student ranking program. A student has 4 subject grades, which are Arabic, math, English and computer. We will enter each subject grade of each student into your program, and your program should be able to sort these students according to the total points they get in descending order. The skeleton code should be like the following:

```
typedef struct students{
    char name[30];
    int arabic, math, english, computer;
    int total;
} Student;
void main()
{
    Student people[5];
    ...
}
```

Your program should have the following input & output.

Enter how many students> 4

__ Student 1's name> Ali

__ Arabic> 62

__ Math> 99

__ English> 63

__ Computer> 87

__ Student 2's name> Ahmed

__ Arabic> 98

__ Math> 87

__ English> 84

__ Computer> 99

__ Student 3's name> Mona

__ Arabic> 92

__ Math> 60

__ English> 58

__ Computer> 62

__ Student 4's name> Heba

__ Arabic> 78

__ Math> 72

__ English> 88

__ Computer> 76

=====

Rank	Arb	Math	Eng	Com	Total	Name
1	98	87	84	99	368	Ahmed
2	78	72	88	76	314	Heba
3	62	99	63	87	311	Ali
4	92	60	58	62	272	Mona



9 Pointers and Memory Allocation

Pointers are often thought to be the most difficult aspect of C. It's true that many people have various problems with pointers and that many programs founder on pointer-related bugs. Actually, though, many of the problems are not so much with the pointers themselves but rather with the memory they point to, and more specifically, when there isn't any valid memory which they point to. As long as you're careful to ensure that the pointers in your programs always point to valid memory, pointers can be useful, powerful, and relatively trouble-free tools.

A **pointer** is a variable that points at, or refers to, another variable. That is, if we have a pointer variable of type “pointer to `int`”, it might point to the `int` variable `i`, or to the third cell of the `int` array `a`. Given a pointer variable, we can ask questions like, “What's the value of the variable that this pointer points to?”

Why would we want to have a variable that refers to another variable? Why not just use that other variable directly? The answer is that a level of **indirection** can be very useful. Indirection is just another word for the situation when one variable refers to another. In general pointers can be used to:

- manipulate arrays more easily by moving pointers to them (or to parts of them), instead of moving the arrays themselves
- Call by reference instead of call by value while calling functions.
- return more than one value from a function
- communicate information about memory, as in the function `malloc()` and the operator `new`, which returns the location of free memory by using a pointer
- pointer notation compiles into **faster, more efficient** code than, for example, array notation

9.1 Basic Pointer Operations

The first things to do with pointers are to declare a pointer variable, set it to point somewhere, and finally manipulate the value that it points to. A simple pointer declaration looks like this:

```
int *ip;
```

This declaration looks like our earlier declarations, with one obvious difference: that **asterisk**. The asterisk means that **ip**, the variable we're declaring, is not of type **int**, but rather of type **pointer-to-int**. We may think of setting a pointer variable to point to another variable as a two-step process:

- 1) Generate a pointer to that other variable,
- 2) Assign this new pointer to the pointer variable. We can say that a pointer variable has a value, and that its value is “pointer to that other variable (memory address)”. This will make more sense when we see how to generate pointer values.

Pointers (that is, pointer values) are generated with the “**address-of**” operator **&**, which we can also think of as the “**pointer-to**” **operator**. We demonstrate this by declaring (and initializing) an **int** variable **i**, and then setting **ip** to point to it:

```
int i = 5;  
ip = &i;
```

The assignment expression **ip = &i;** contains both parts of the “two-step process”: **&i** generates a pointer to **i**, and the assignment operator assigns the new pointer to the variable **ip**. Now **ip** “**points to**” **i** as we can illustrate with this picture:



i is a variable of type **int**, so the value in its box is a number, 5. **ip** is a variable of type **pointer-to-int**, so the “value” in its box is an arrow pointing at another box. Referring once again back to the “two-step

process” for setting a pointer variable: the **&** operator draws us the arrowhead pointing at **i**'s box, and the assignment operator **=**, with the pointer variable **ip** on its left, anchors the other end of the arrow in **ip**'s box.

We discover the value pointed to by a pointer using the “**contents-of**” operator, ***** placed in front of a pointer, the ***** operator accesses the value pointed to by that pointer. In other words, if **ip** is a pointer, then the expression ***ip** gives us whatever it is that's in the variable or location pointed to by **ip**. For example, we could write something like

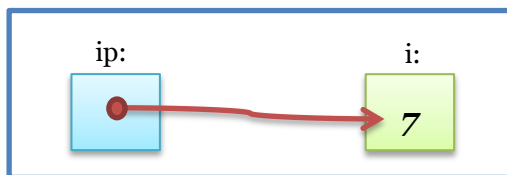
```
printf("%d\n", *ip);
```

which would print 5, since **ip** points to **i**, and **i** is (at the moment) 5.

The contents-of operator ***** does not merely fetch values through pointers; it can also set values through pointers. We can write something like

```
*ip = 7;
```

that means “set whatever **ip** points to to 7”. Again, the ***** tells us to go to the location pointed to by **ip**, but this time, the location isn't the one to fetch from--we're on the left-hand sign of an assignment operator, so ***ip** tells us the location to store to. The result of the assignment ***ip = 7** is that **i**'s value is changed to 7, and the picture changes to:



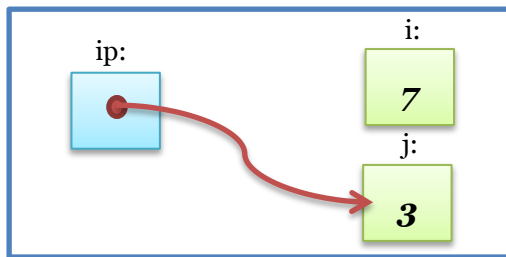
At this point, you may be wondering why we're going through this hassle. If we wanted to set **i** to 7, why didn't we do it directly? We'll begin to explore that next, but first let's notice the difference between changing a pointer (that is, changing what variable it points to) and changing the value at the location it points to. When we wrote ***ip = 7**, we changed the value pointed to by **ip**, but if we declare another variable **j**:

```
int j = 3;
```

and write

```
ip = &j;
```

we've changed **ip** itself. The picture now looks like this:



We have to be careful when we say that a pointer assignment changes “what the pointer points to.” Our earlier assignment

```
*ip = 7;
```

changed the value pointed to by **ip**, but this more recent assignment

```
ip = &j;
```

has changed what variable **ip** points to. If we again call:

```
printf("%d\n", *ip);
```

this time it will print 3 (**j**'s value).

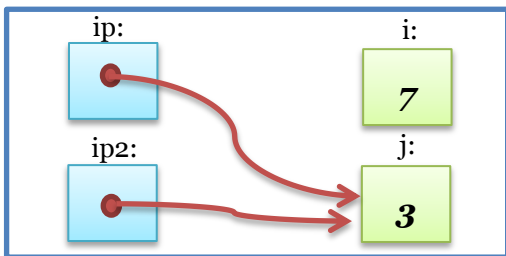
We can also assign pointer values to other pointer variables. If we declare a second pointer variable:

```
int *ip2;
```

then we can say

```
ip2 = ip;
```

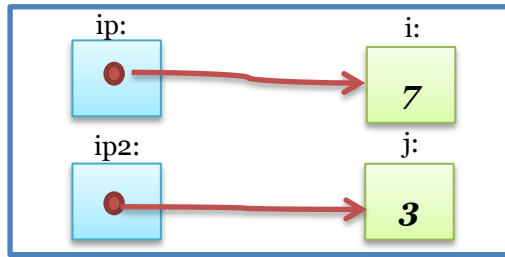
Now **ip2** points where **ip** does; we've essentially made a “*copy*” of the arrow:



Now, if we set **ip** to point back to **i** again:

```
ip = &i;
```

the two arrows point to different places:



We can now see that the two assignments

```
ip2 = ip;
```

and

```
*ip2 = *ip;
```

do two very different things. The first would make **ip2** again point to where **ip** points (in other words, back to **i** again). The second would store, at the location pointed to by **ip2**, a copy of the value pointed to by **ip**; in other words (if **ip** and **ip2** still point to **i** and **j** respectively) it would set **j** to **i**'s value (7).



It's important to keep very clear in your mind the distinction between a pointer and what it points to. The two are like oil and water; you can't mix them. You can't “set **ip** to 5” by writing something like:

```
ip = 5; /* WRONG */
```

5 is an integer, but **ip** is a pointer. You probably wanted to “set the value pointed to by **ip** to 5”, that you express by writing

```
*ip = 5;
```

Similarly, you can't “see what **ip** is” by writing

```
printf("%d\n", ip); /* WRONG */
```

Again, **ip** is a pointer-to-int, but **%d** expects an **int**. To print what **ip** points to, use

```
printf("%d\n", *ip);
```

Finally, a few more notes about pointer declarations. The `*` in a pointer declaration is related to, but different from, the contents-of operator `*`. After we declare a pointer variable:

```
int *ip;
```

the expression

```
ip = &i
```

sets what `ip` points to (that is, which location it points to), while the expression

```
*ip = 5
```

sets the value of the location pointed to by `ip`. On the other hand, if we declare a pointer variable and include an initializer:

```
int *ip3 = &i;
```

we're setting the initial value for `ip3`, which is where `ip3` will point, so that initial value is a pointer. (In other words, the `*` in the declaration `int *ip3 = &i;` is not the contents-of operator, it's the indicator that `ip3` is a pointer.)

If you have a pointer declaration containing an initialization, and you ever have occasion to break it up into a simple declaration and a conventional assignment, do it like this:

```
int *ip3;  
ip3 = &i;
```

Don't write

```
int *ip3;  
*ip3 = &i;
```

or you'll be trying to mix oil and water again. Also, when we write

```
int *ip;
```

although the asterisk affects `ip`'s type, it goes with the identifier name `ip`, not with the type `int` on the left. To declare two pointers at once, the declaration looks like

```
int *ip1, *ip2;
```

Some people write pointer declarations like this:

```
int* ip;
```

This works for one pointer, because C essentially ignores whitespace. But if you ever write

```
int* ip1, ip2;          /* PROBABLY WRONG */
```

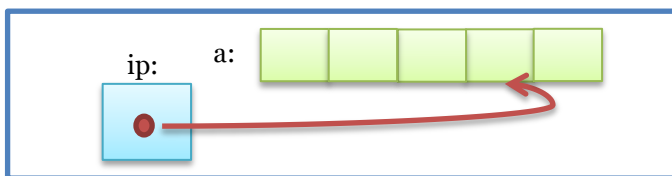
it will declare one pointer-to-int **ip1** and one plain int **ip2**, which is probably not what you meant.

9.1.1 Pointers and Arrays: Pointer Arithmetic

Pointers do not have to point to single variables. They can also point at the cells of an array. For example, we can write:

```
int *ip;  
int a[10];  
ip = &a[3];
```

and we would end up with **ip** pointing at the fourth cell of the array **a** (remember, arrays are 0-based, so **a[0]** is the first cell). We could illustrate the situation like this:



We'd use this **ip** just like the one in the previous section: ***ip** gives us what **ip** points to, which in this case will be the value in **a[3]**.

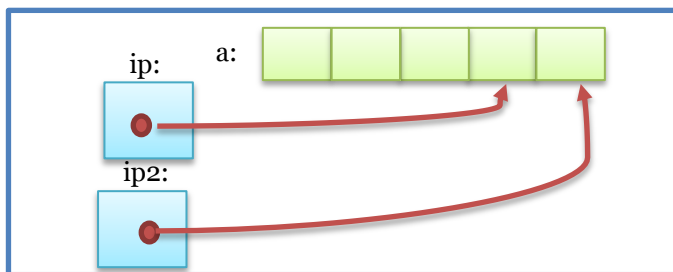
Once we have a pointer pointing into an array, we can start doing **pointer arithmetic**. Given that **ip** is a pointer to **a[3]**, we can add 1 to **ip**:

```
ip + 1
```

What does it mean to add one to a pointer? In C, it gives a pointer to the cell one farther on, which in this case is **a[4]**. To make this clear, let's assign this new pointer to another pointer variable:

```
ip2 = ip + 1;
```

Now the picture looks like this:



If we now do

```
*ip2 = 4;
```

we've set **a[4]** to **4**. But it's not necessary to assign a new pointer value to a pointer variable in order to use it; we could also compute a new pointer value and use it immediately:

```
*(ip + 1) = 5;
```

In this last example, we've changed **a[4]** again, setting it to **5**. The parentheses are needed because the unary “contents of” operator ***** has higher precedence (i.e., binds more tightly than) the addition operator. If we wrote ***ip + 1**, without the parentheses, we'd be fetching the value pointed to by **ip**, and adding **1** to that value. The expression ***(ip + 1)**, on the other hand, accesses the value one past the one pointed to by **ip**.

Given that we can add **1** to a pointer, it's not surprising that we can add and subtract other numbers as well. If **ip** still points to **a[3]**, then

```
*(ip + 3) = 7;
```

sets **a[6]** to **7**, and

```
*(ip - 2) = 4;
```

sets **a[1]** to **4**.

Up above, we added **1** to **ip** and assigned the new pointer to **ip2**, but there's no reason we can't add one to a pointer, and change the same pointer:

```
ip = ip + 1;
```

Now **ip** points one past where it used to (to **a[4]**, if we hadn't changed it in the meantime). The shortcuts we learned in a previous chapter all work for pointers, too: we could also increment a pointer using:

```
ip += 1;
```

or

```
ip++;
```

Of course, pointers are not limited to **ints**. It's quite common to use pointers to other types, especially **char**. Here is the innards of the **mystrcmp** function we saw in a previous chapter, rewritten to use pointers. (**mystrcmp**, you may recall, compares two strings, character by character.)

```

char *p1 = &str1[0], *p2 = &str2[0];

while(1){
    if(*p1 != *p2)
        return *p1 - *p2;
    if(*p1 == '\0' && *p2 == '\0')
        return 0;
    p1++;
    p2++;
}

```

The auto-increment operator `++` makes it easy to do two things at once. We've seen idioms like `a[i++]` which accesses `a[i]` and simultaneously increments `i`, leaving it referencing the next cell of the array `a`. We can do the same thing with pointers: an expression like `*ip++` lets us access what `ip` points to, while simultaneously incrementing `ip` so that it points to the next element. The pre-increment form works, too: `*++ip` increments `ip`, then accesses what it points to. Similarly, we can use notations like `*ip--` and `*--ip`.

One question that comes up is whether the expression `*p++` increments `p` or what it points to. The answer is that it increments `p`. To increment what `p` points to, you can use `(*p)++`.



When you're doing pointer arithmetic, you have to remember how big the array the pointer points into is, so that you don't ever point outside it. If the array `a` has 10 elements, you can't access `a[50]` or `a[-1]` or even `a[10]` (remember, the valid index for a 10-element array run from 0 to 9). Similarly, if `a` has 10 elements and `ip` points to `a[3]`, you can't compute or access `ip + 10` or `ip - 5`.

9.1.2 Pointer Subtraction and Comparison

As we've seen, you can add an integer to a pointer to get a new pointer, pointing somewhere beyond the original (as long as it's in the same array). For example, you might write:

```
ip2 = ip1 + 3;
```

Applying a little algebra, you might wonder whether

```
ip2 - ip1 = 3
```

and the answer is, yes. When you subtract two pointers, as long as they point into the same array, the result is the number of elements separating them. You can also ask (again, as long as they point into the same array) whether one pointer is greater or less than another: one pointer is “greater than” another if it points beyond where the other one points. You can also compare pointers for equality and inequality: two pointers are equal if they point to the same variable or to the same cell in an array, and are (obviously) unequal if they don't. (When testing for equality or inequality, the two pointers do not have to point into the same array)

One common use of pointer comparisons is when copying arrays using pointers. Here is a code fragment which copies 10 elements from **array1** to **array2**, using pointers. It uses an end pointer, **ep**, to keep track of when it should stop copying.

```
int array1[10], array2[10];
int *ip1, *ip2 = &array2[0];
int *ep = &array1[10];
for(ip1 = &array1[0]; ip1 < ep; ip1++)
    *ip2++ = *ip1;
```

As we mentioned, there is no element **array1[10]**, but it is legal to compute a pointer to this (nonexistent) element, as long as we only use it in pointer comparisons like this (that is, as long as we never try to fetch or store the value that it points to.)

9.1.3 Null Pointers

We said that the value of a pointer variable is a pointer to some other variable. There is one other value a pointer may have: it may be set to a **null pointer**. A null pointer is a special pointer value that is known not to point anywhere. What this means that no other valid pointer, to any other variable or array cell or anything else, will ever compare equal to a null pointer.

The most straightforward way to “get” a null pointer in your program is by using the predefined constant **NULL**, which is defined for you by several standard header files, including **<stdio.h>**, **<stdlib.h>**, and **<string.h>**. To initialize a pointer to a null pointer, you might use code like

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

```
int *ip = NULL;
```

and to test it for a null pointer before inspecting the value pointed to you might use code like

```
if(ip != NULL)
    printf("%d\n", *ip);
```

It is also possible to refer to the null pointer by using a constant **0**, and you will see some code that sets null pointers by simply doing

```
int *ip = 0;
```

Furthermore, since the definition of “true” in C is a value that is not equal to **0**, you will see code that tests for non-null pointers with abbreviated code like

```
if(ip)
    printf("%d\n", *ip);
```

This has the same meaning as our previous example; **if(ip)** is equivalent to **if(ip != 0)** and to **if(ip != NULL)**. All of these uses are legal, and although I recommend that you use the constant **NULL** for clarity, you will come across the other forms, so you should be able to recognize them.

You can use a **null** pointer as a placeholder to remind yourself (or, more importantly, to help your program remember) that a pointer variable does not point anywhere at the moment and that you should not use the “contents of” operator on it (that is, you should not try to inspect what it points to, since it doesn't point to anything). A function

that returns pointer values can return a **null** pointer when it is unable to perform its task.

In general, C does not initialize pointers to null for you, and it never tests pointers to see if they are null before using them. If one of the pointers in your programs points somewhere some of the time but not all of the time, an excellent convention to use is to set it to a null pointer when it doesn't point anywhere valid, and to test to see if it's a null pointer before using it. But you must use explicit code to set it to **NULL**, and to test it against **NULL**.

9.2 Pointers and Passing Arguments

Functions usually return only one value and when arguments are **passed by value**, the called function cannot alter the values passed and have those changes reflected in the calling function. Pointers allow the programmer to "return" more than one value by allowing the arguments to be **passed by reference** which allows the function to alter the values pointed to and thus "return" more than one value from a function.

To explain the concept of calling by reference in C, we consider the program shown in Listing 9-1 that tries to swap the value of two variables.

Listing 9-1: Wrong Swap Program

```
1. #include <stdio.h>
2. void swap(int,int);
3. int main(){
4.     int x = 4, y = 10;
5.     swap(x,y);
6.     printf("X=%d Y=%d",x,y);
7.     return 0;
8. }

9. void swap(int a,int b){
10.    int temp;
11.    temp = a;
12.    a = b;
13.    b = temp;
14. }
```

The result of running the above example would be:

X=4 Y=10

The result prints the original values stored in **x** and **y**. The variables **a**, **b**, and **temp** are local to **swap()** and their values have no effect on the original variables.

As presented in Listing 9-2, with the use of pointers and passing by reference, the function can affect the original variables and thus perform the required function and swap the values of the two variables.

Listing 9-2: Correct Swap Program

```
1. #include <stdio.h>
2. void swap(int *, int *);
3. int main(){
4.     int x = 4, y = 10;
5.     swap(&x, &y);
6.     printf("X=%d Y=%d",x,y);
7.     return 0;
8. }

9. void swap(int *a, int *b){
10.    int temp;
11.    temp = *a;
12.    *a = *b;
13.    *b = temp;
14. }
```

The result of running the above example would be:

X=10 Y=4

The values have been exchanged by the function **swap()**. Within the **main()** function the **&** operator causes the **address** (reference) of arguments **x** and **y** to be passed in the call to **swap()**. In the **swap()** function header, the addresses being passed from the calling function are received in pointer type variables (**int *a**, **int *b**). Within the **swap()** function body, the ***** operator is used to retrieve values held at

the addresses that were passed. When, the values of pointers are changed, the values in memory location also changed correspondingly. Hence, change made to ***a** and ***b** was reflected in **x** and **y** in main function. This technique is known as **call by reference** in C programming.

9.3 Memory Allocation

Another very important usage of pointers is dynamic memory allocation. In this section you will meet the **malloc**, C's dynamic memory allocation function. You have to be careful while dealing with dynamic memory allocation. **malloc** operates at a pretty "**low level**"; you will often find yourself having to do a certain amount of work to manage the memory it gives you. If you don't keep accurate track of the memory which **malloc** has given you, and the pointers of yours which point to it, it's all too easy to accidentally use a pointer which points "nowhere", with generally unpleasant results. (The basic problem is that if you assign a value to the location pointed to by a pointer:

```
*p = 0;
```

and if the pointer **p** points "somewhere" that may be in use by some other part of your program, or even worse, if the operating system has not protected itself from you and "somewhere" is in fact in use by the operating system, things could get ugly.

9.3.1 Allocating Memory with malloc

A problem with many simple programs is that they tend to use **fixed-size arrays** that may or may not be big enough. We have an array of 100 **ints** for the numbers which the user enters and wishes to find the average of--what if the user enters 101 numbers? We have an array of 100 **chars** which we pass to **getline** to receive the user's input--what if the user types a line of 200 characters? If we're lucky, the relevant parts of the program check how much of an array they've used, and print an error message or otherwise gracefully abort before overflowing the array. If we're not so lucky, a program may sail off the end of an array, overwriting other data and behaving quite badly. In

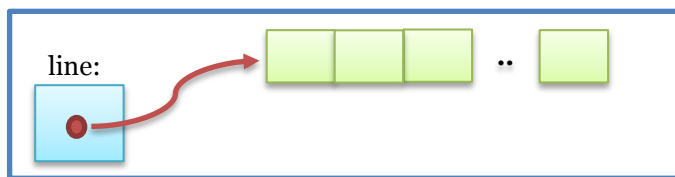
either case, the user doesn't get his job done. How can we avoid the restrictions of fixed-size arrays?

The answers all involve the standard library function `malloc`. Very simply, `malloc` returns a pointer to *n bytes of memory* which we can do anything we want to with. If we didn't want to read a line of input into a fixed-size array, we could use `malloc`, instead. Here's the first step:

```
#include <stdlib.h>

char *line;
int linelen = 100;
line = malloc(linelen);
/* incomplete -- malloc's return value not checked */
getline(line, linelen);
```

`malloc` is declared in `<stdlib.h>`, so we `#include` that header in any program that calls `malloc`. A “byte” in C is, by definition, an amount of storage suitable for storing one character, so the above invocation of `malloc` gives us exactly as many chars as we ask for. We could illustrate the resulting pointer like this:



The 100 bytes of memory (not all of which are shown) pointed to by `line` are those allocated by `malloc`. They are brand-new memory, conceptually a bit different from the memory which the compiler arranges to have allocated automatically for our conventional variables. The 100 boxes in the figure don't have a name next to them, because they're not storage for a variable we've declared.

As a second example, we might have occasion to allocate a piece of memory, and to copy a string into it with `strcpy`:

```
char *p = malloc(15);
/* incomplete -- malloc's return value not checked */
strcpy(p, "Hello, world!");
```

When copying strings, remember that all strings have a terminating `\0` character. If you use `strlen` to count the characters in a string for you, that count will not include the trailing `\0`, so you must add one before calling `malloc`:

```
char *somestring, *copy;
...
copy = malloc(strlen(somestring) + 1);    /* +1 for \0 */
/* incomplete -- malloc's return value not checked */
strcpy(copy, somestring);
```



What if we're not allocating characters, but integers? If we want to allocate 100 `ints`, how many bytes is that? If we know how big `ints` are on our machine (i.e. depending on whether we're using a 16- or 32-bit machine) we could try to compute it ourselves, but it's much safer and more portable to let C compute it for us. C has a `sizeof` operator, which computes the size, in bytes, of a variable or type. It's just what we need when calling `malloc`. To allocate space for 100 `ints`, we could call:

```
int *ip = malloc(100 * sizeof(int));
```

The use of the `sizeof` operator tends to look like a function call, but it's really an operator, and it does its work at compile time. Since we can use array indexing syntax on pointers, we can treat a pointer variable after a call to `malloc` almost exactly as if it were an array. In particular, after the above call to `malloc` initializes `ip` to point at storage for 100 `ints`, we can access `ip[0]`, `ip[1]`, ... up to `ip[99]`. This way, we can get the effect of an array even if we don't know until run time how big the “array” should be.

Our examples so far have all had a significant omission: they have not checked `malloc`'s return value. Obviously, no real computer has an infinite amount of memory available, so there is no guarantee that `malloc` will be able to give us as much memory as we ask for. If we call

`malloc(100000000)`, or if we call `malloc(10)` 10,000,000 times, we're probably going to run out of memory.

When `malloc` is unable to allocate the requested memory, it returns a null pointer. A null pointer, remember, points definitively nowhere. It's a “***not a pointer***” marker; it's not a pointer you can use. (As we said before, a null pointer can be used as a failure return from a function that returns pointers, and `malloc` is a perfect example.) Therefore, whenever you call `malloc`, it's vital to check the returned pointer before using it! If you call `malloc`, and it returns a null pointer, and you go off and use that null pointer as if it pointed somewhere, your program probably won't last long. Instead, a program should immediately check for a null pointer, and if it receives one, it should at the very least print an error message and exit, or perhaps figure out some way of proceeding without the memory it asked for. But it cannot go on to use the null pointer it got back from `malloc` in any way, because that null pointer by definition points nowhere.

A call to `malloc`, with an error check, typically looks something like this:

```
int *ip = malloc(100 * sizeof(int));
if(ip == NULL){
    printf("out of memory\n");
    return -1;          //or exit(-1)
}
```

After printing the error message, this code should return to its caller, or exit from the program entirely; it cannot proceed with the code that would have used `ip`. As you notice here, the return statement use the (-1) value to send a message the operating system that this program termination is abnormal one (due to an internal error).

Of course, in our examples so far, we've still limited ourselves to “fixed size” regions of memory, because we've been calling `malloc` with fixed arguments like 10 or 100. However, since the sizes are now values which can in principle be determined at run-time, we've at least moved beyond having to recompile the program (with a bigger array)

to accommodate longer lines, and with a little more work, we could arrange that the “arrays” automatically grew to be as large as required.

9.3.2 Freeing Memory

Memory allocated with `malloc` lasts as long as you want it to. It does not automatically disappear when a function returns, as automatic-duration variables do, but it does not have to remain for the entire duration of your program, either. Just as you can use `malloc` to control exactly when and how much memory you allocate, you can also control exactly when you *de-allocate* it.

In fact, many programs use memory on a transient basis. They allocate some memory, use it for a while, but then reach a point where they don't need that particular piece any more. Because memory is not inexhaustible, it's a good idea to de-allocate (that is, release or free) memory you're no longer using.

Dynamically allocated memory is de-allocated with the `free` function. If `p` contains a pointer previously returned by `malloc`, you can call:

```
free(p);
```

which will “give the memory back” to the stock of memory (sometimes called the “pool”) from which `malloc` requests are satisfied. Calling `free` is sort of the ultimate in recycling: it costs you almost nothing, and the memory you give back is immediately usable by other parts of your program or other programs. Freeing unused memory is a good idea, but it's not mandatory. When your program exits, any memory which it has allocated but not freed should be automatically released.

Naturally, once you've freed some memory you must remember not to use it any more. After calling:

```
free(p);
```

it is probably the case that `p` still points at the same memory. However, since we've given it back, it's now not available, and a later call to `malloc` might give that memory to some other part of your program. If the variable `p` is a global variable or will otherwise stick around for a

while, one good way to record the fact that it's not to be used any more would be to set it to a null pointer:

```
free(p);  
p = NULL;
```

Now we don't even have the pointer to the freed memory any more, and (as long as we check to see that **p** is non-NULL before using it), we won't misuse any memory via the pointer **p**.



When thinking about **malloc**, **free**, and dynamically-allocated memory in general, remember again the distinction between a pointer and what it points to. If you call **malloc** to allocate some memory, and store the pointer which **malloc** gives you in a local pointer variable, what happens when the function containing the local pointer variable returns? If the local pointer variable has automatic duration (which is the default, unless the variable is declared static), it will disappear when the function returns. But for the pointer variable to disappear says nothing about the memory pointed to! That memory still exists and, as far as **malloc** and **free** are concerned, is still allocated. The only thing that has disappeared is the pointer variable you had which pointed at the allocated memory. Furthermore, if it contained the only copy of the pointer you had, once it disappears, you'll have no way of freeing the memory, and no way of using it, either. Using memory and freeing memory both require that you have at least one pointer to the memory!

9.3.3 Reallocating Memory Blocks

Sometimes you're not sure at first how much memory you'll need. For example, if you need to store a series of items you read from the user, and if the only way to know how many there are is to read them until the user types some “end” signal, you'll have no way of knowing, as you begin reading and storing the first few, how many you'll have seen by the time you do see that “end” marker. You might want to allocate room for, say, 100 items, and if the user enters a 101st item before entering the “end” marker, you might wish for a way to say “uh, **malloc**, remember those 100 items I asked for? Could I change my mind and have 200 instead?”

In fact, you can do exactly this, with the **realloc** function. You hand **realloc** an old pointer (such as you received from an initial call to

`malloc`) and a new size, and `realloc` does what it can to give you a chunk of memory big enough to hold the new size. For example, if we wanted the `ip` variable from an earlier example to point at 200 ints instead of 100, we could try calling

```
ip = realloc(ip, 200 * sizeof(int));
```

You and `realloc` have to worry about the case where `realloc` can't make the old block of memory bigger but rather has to relocate it elsewhere in order to find enough contiguous space for the new requested size. `realloc` does this by returning a new pointer. If `realloc` was able to make the old block of memory bigger, it returns the same pointer. If `realloc` has to go elsewhere to get enough contiguous memory, it returns a pointer to the new memory, ***after copying your old data there***. In this case, after it makes the copy, it frees the old block automatically. Finally, if `realloc` can't find enough memory to satisfy the new request at all, it returns a null pointer. Therefore, you usually don't want to overwrite your old pointer with `realloc`'s return value until you've tested it to make sure it's not a null pointer. You might use code like this:

```
int *newp;
newp = realloc(ip, 200 * sizeof(int));
if(newp != NULL)
    ip = newp;
else{
    printf("out of memory\n");
    /* exit or return */
    /* but ip still points at 100 ints */
}
```

If `realloc` returns something other than a null pointer, it succeeded, and we set `ip` to what it returned. (We've either set `ip` to what it used to be or to a new pointer, but in either case, it points to where our data is now.) If `realloc` returns a null pointer, however, we hang on to our old pointer in `ip` which still points at our original 100 values.

Putting this all together, Listing 9-3 introduces a piece of code that reads lines of text from the user, treats each line as an integer by calling `atoi`, and stores each integer in a dynamically-allocated array:

We use two different variables to keep track of the array pointed to by `ip`. `nalloc` is how many elements we've allocated, and `nitems` is how many of them are in use. Whenever we're about to store another item in the array, if `nitems >= nalloc`, the old array is full, and it's time to call `realloc` to make it bigger.

Listing 9-3: Read Lines from a User

```
1. #define MAXLINE 100

2. char line[MAXLINE];
3. int *ip;
4. int nalloc, nitems;

5. nalloc = 100;
6. ip = malloc(nalloc * sizeof(int));

7. if(ip == NULL){
8.     printf("out of memory\n");
9.     exit(1);
10. }

11. nitems = 0;

12. while(getline(line, MAXLINE) != EOF){
13.     if(nitems >= nalloc){        /* increase allocation */
14.         int *newp;
15.         nalloc += 100;
16.         newp = realloc(ip, nalloc * sizeof(int));
17.         if(newp == NULL){
18.             printf("out of memory\n");
19.             exit(1);
20.         }
21.         ip = newp;
22.     }

23.     ip[nitems++] = atoi(line);
24. }
```

9.3.4 Dynamic Memory Allocation in C++

Management of dynamic memory in C++ is quite similar to C in most respects. Although the library functions are likely to be available, C++ has two additional operators – **new** and **delete** – which enable code to be written more clearly, succinctly and flexibly, with less likelihood of errors. The **new** operator can be used in three ways:

```
p_var = new typename;  
p_var = new type(initializer);  
p_array = new type [size];
```

In the first two cases, space for a single object is allocated; the second one includes **initialization**. The third case is the mechanism for allocating space for an array of objects.

The **delete** operator can be invoked in two ways:

```
delete p_var;  
delete[] p_array;
```

The first is for a single object; the second de-allocates the space used by an array. It is very important to use the correct de-allocator in each case.

There is no operator that provides the functionality of the C **realloc()** function.

Here is the code to dynamically allocate an array and initialize the fourth element:

```
int* pointer;  
pointer = new int[10];  
pointer[3] = 99;
```

Using the array access notation is natural. De-allocation is performed thus:

```
delete[] pointer;  
pointer = NULL;
```

Again, assigning NULL to the pointer after de-allocation is just good programming practice.

9.3.5 Pointer Safety

At the beginning of the chapter, we said that the hard thing about pointers is not so much manipulating them as ensuring that the memory they point to is valid. When a pointer doesn't point where you think it does, if you inadvertently access or modify the memory it points to, you can damage other parts of your program or (in some cases) other programs or the operating system itself!

When we use pointers to simple variables, there's not much that can go wrong. When we use pointers into arrays, and begin moving the pointers around, we have to be more careful, to ensure that the roving pointers always stay within the bounds of the array(s). When we begin passing pointers to functions, and especially when we begin returning them from functions we have to be more careful still, because the code using the pointer may be far removed from the code which owns or allocated the memory.

One particular problem concerns functions that return pointers. Where is the memory to which the returned pointer points? Is it still around by the time the function returns? One thing a function must not do is to return a pointer to one of its own, local, automatic-duration arrays. Remember that automatic-duration variables (which includes all non-static local variables), including automatic-duration arrays, are de-allocated and disappear when the function returns. If a function returns a pointer to a local array, that pointer will be invalid by the time the caller tries to use it.

Finally, when we're doing dynamic memory allocation with **malloc**, **realloc**, and **free**, we have to be most careful of all. Dynamic allocation gives us a lot more flexibility in how our programs use memory, although with that flexibility comes the responsibility that we manage dynamically allocated memory carefully. The possibilities for misdirected pointers and associated mayhem are greatest in programs that make heavy use of dynamic memory allocation. You can reduce these possibilities by designing your program in such a way that it's

easy to ensure that pointers are used correctly and that memory is always allocated and de-allocated correctly.

9.4 Exercises

- 1) In which header file is the NULL macro defined?
- a. `stdio.h`
 - b. `stddef.h`
 - c. `stdio.h` and `stddef.h`
 - d. `math.h`
- 2) Which pair of the following statements are equivalent?

- a. `*value[1];`
`*(value + 1);`
- b. `**value;`
`*value;`
- c. `*value[2];`
`(*value++)++;`
- d. `*value;`
`&value;`

- 3) What is the output of this C code?

```
#include <stdio.h>
void foo(int*);
int main()
{
    int i = 10, *p = &i;
    foo(p++);
}
void foo(int *p)
{
    printf("%d\n", *p);
}
```

- a. 10
 - b. Some garbage value
 - c. Compile time error
 - d. segmentation fault.
- 4) If we say
- ```
int i = 5;
int *ip = &i;
```
- then what is ip? What is its value?

- 5) If `ip` is a pointer to an integer, what does `ip++` mean?  
What does  
    `*ip++ = 0;`  
do?
- 6) How much memory does the call `malloc(10)` allocate?  
What if you want enough memory for 10 ints?
- 7) If `p` is a pointer, what does `p[i]` mean?

- 8) Run the following program, show the output and answer the questions.

```
#include <stdio.h>
void main(void){
 char a, *pa; // Statement 1
 pa = &a; // Statement 2
 printf("pa = &a --> pa = %p \n", pa);
 pa = pa + 1; // Statement 3
 printf("pa = pa + 1 --> pa = %p \n", pa);
 pa = pa + 3; // Statement 4
 printf("pa = pa + 3 --> pa = %p \n", pa);
 pa = pa - 1; // Statement 5
 printf("pa = pa - 1 --> pa = %p \n", pa);
}
```

- a. Can we add an integer to a variable such as `pa` that stores addresses?
- b. Can we subtract integers from a variable such as `pa` that store addresses?
- c. Can we multiply integers to a variable such as `pa` that stores addresses as shown below? What error do you get?  
    `pa = pa * 3;`
- d. What was the first address stored in `pa` in Statement 3?
- e. After 1 was added to `pa` in Statement 3, what address was stored in `pa`?
- f. After 3 was added to `pa` in Statement 4, what address was stored in `pa`?
- g. After 1 was subtracted from `pa` in Statement 5, what address was stored in `pa`?
- h. Now change only Statement 1 to the following statement:  
    `int a, *pa;`

- i. Rerun the program and answer the same questions, namely:
  - a) What was the first address stored in pa in Statement 3?
  - b) After 1 was added to pa in Statement 3, what address was stored in pa?
  - c) After 3 was added to pa in Statement 4, what address was stored in pa?
  - d) After 1 was subtracted from pa in Statement 5, what address was stored in pa?
- j. Adding 1 to a variable that holds character addresses adds what number to the address?
- k. Adding 1 to a variable that holds integer addresses adds what number to that address?
- l. Since the answer to question j is 1, characters are stored in memory using only 1 byte. A byte is a unit of measure of memory. How many bytes are used to store integers on your computer? Take note that the number of bytes used to store characters, integers and floats varies depending on the type of computer or platform (e.g. 32 bits, 64 bits system etc.) and/or the target platform of your program. You can use the sizeof() function to check the size of variables as shown in the following program example.

```
#include <stdio.h>
void main(void)
{
 char a = 'W';
 int b = 100;
 float c = 1.234;
 double d = 100000.34;
 printf("Size of a = %d byte(s).\n", sizeof(a));
 printf("Size of b = %d byte(s).\n", sizeof(b));
 printf("Size of c = %d byte(s).\n", sizeof(c));
 printf("Size of d = %d byte(s).\n", sizeof(d));
}
```

- 9) Run the following program, show the output and answer the questions.

```
#include <stdio.h>
void main(void)
{
 float a[4], *b, c;
 b = &c; // Statement 1
 printf("b = %p\n", b);
}
```



a. Did Statement 1 execute or did it give an error?

b. Change Statement 1 to the following:

```
a = &c;
```

Try running it. Did it give an error? What was that error?

As a conclusion, both a and b store addresses, that is they are both pointers. However, the address stored in b (a pointer) can be changed, it is a pointer variable. The address of a (an array) cannot be changed, it is a pointer constant.

10) For the following questions, use the declaration of variables shown below. You can try building a simple program using the given specification in order to answer the questions.

```
int i, j[5] = {4, 5, 6, 7, 8}, *ptr1 = &j[0],
*ptr3;
float x[5] = {4.0, 5.0, 6.0, 7.0, 8.0}, *ptr2;
```

For each statement below, specify which are valid and which aren't.

- a. `ptr1 = ptr1 + 3;`
- b. `j = j + 1;`
- c. `ptr1 = j + 1;`
- d. `ptr2 = ptr1;`
- e. `ptr1 = j[1];`
- f. `ptr1 = 2;`
- g. `i = ptr1;`
- h. `ptr3 = ptr1;`
- i. `i = j[2];`
- j. `ptr2 = x;`
- k. `ptr1 = ptr1[2];`
- l. `x = &ptr2[2];`
- m. `j = ptr1 + 3;`
- n. `ptr1 = &j[1];`

## **APPENDIX I: Compilation of a C Program**

This Appendix helps you to get to the point where you can compile, link, run, and debug C/C++ programs. This depends on what operating system you have, so we'll see how to get a C/C++ project up and running under Windows, and Linux.

## Compiling C++ Programs under Windows

This section assumes that you are using Microsoft Visual Studio 2010 (VS2010). If you don't have it, alternatively, you can download Visual C++ 2010 or 2012 Express Edition, a free version of Microsoft's development environment sporting a fully-functional C++ compiler. The express edition of Visual C++ lacks support for advanced Windows development, but is otherwise a perfectly fine C++ compiler. You can get Visual C++ Express Edition from:

[www.microsoft.com/visualstudio/eng/products/](http://www.microsoft.com/visualstudio/eng/products/)

VS2010 organizes C++ code into “projects,” collections of source files that will be built into a program. The first step in creating a C++ program is to get an empty C++ project up and running, then to populate it with the necessary files. To begin, open VS2010 and from the File menu choose **New > Project....** You should see a window as shown in Figure 10-1.

As you can see, VS2010 has template support for all sorts of different projects, most of which are for Microsoft-specific applications such as dynamic-link libraries (DLLs) or ActiveX controls. We're not particularly interested in most of these choices - we just want a simple C++ program! To create one, find and choose Win32 Console Application. Give your project an appropriate name, and then click OK. You should now see a window shown in Figure 10-2, which will ask you to configure project settings:

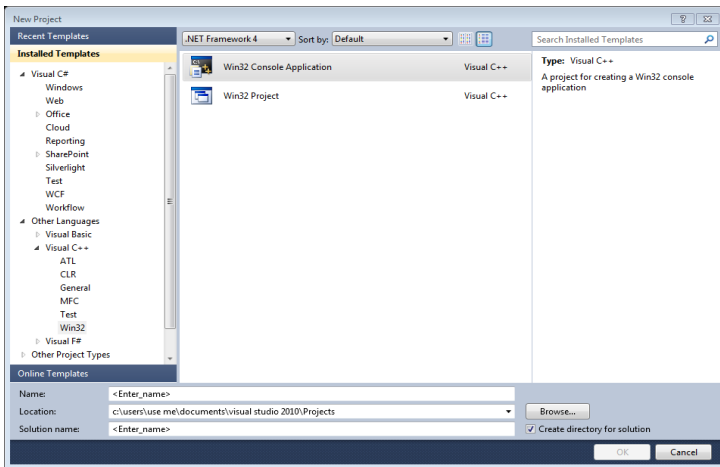


Figure 10-1: Visual Studio New Project

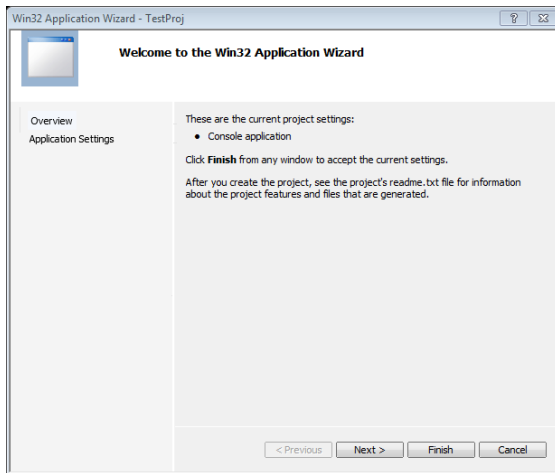
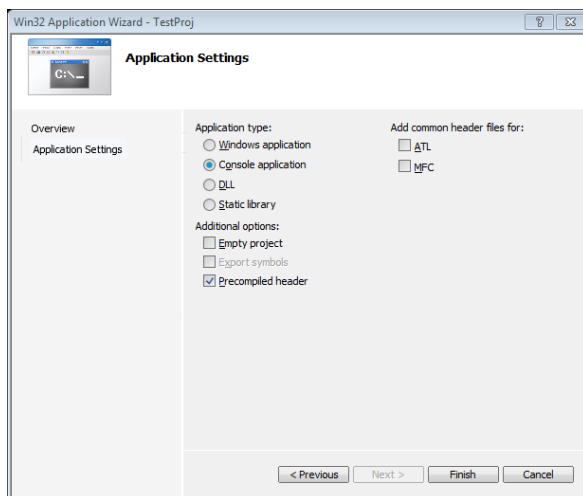


Figure 10-2: Win32 Application Wizard

At this point, hit Next >, and you'll be presented with the screen shown in Figure 10-3. Keep all of the default settings listed here, but make sure that you check the box marked **Empty Project**. Otherwise VS2010 will give you a project with all sorts of Microsoft-specific features built into it. Once you've checked that box, click Finish and you'll have a fully functional C++ project.

Now, it's time to create and add some source files to this project so that you can enter C++ code. To do this, go to **Project > Add New**

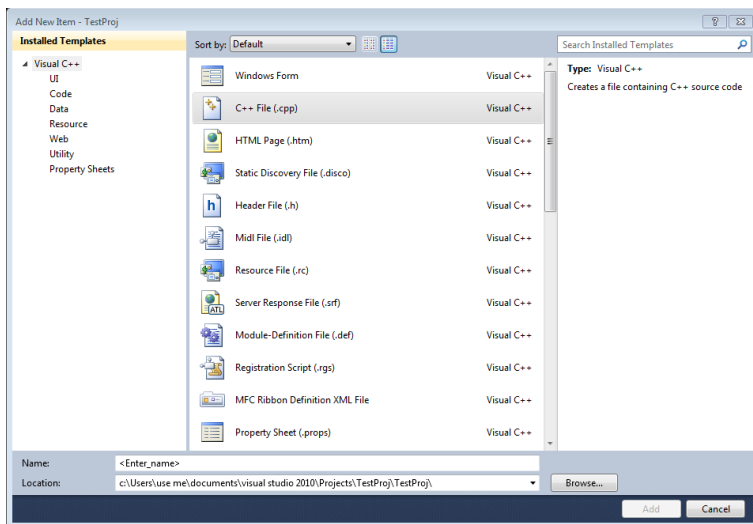
**Item...** (Or press **CTRL+SHIFT+A**). You'll be presented with the dialog box presented in **Figure 10-4**.



**Figure 10-3: Win32 Application Settings**

Choose C++ File (.cpp) and enter a name for it inside the Name field. VS2010 automatically appends .cpp to the end of the filename, so don't worry about manually entering the extension. Once you're ready, click Add and you should have your source file ready to go. Any C++ code you enter in here will be considered by the compiler and built into your final application.

Once you've written the source code, you can compile and run your programs by pressing **F5**, choosing Debug> Start Debugging, or clicking the green “play” icon. By default VS2010 will close the console window after your program finishes running, and if you want the window to persist after the program finishes executing you can run the program without debugging by pressing **CTRL+F5** or choosing Debug > Start Without Debugging. You should be all set to go!



**Figure 10-4: Add New Item Dialog Box**

## Compiling C++ Programs under Linux

For those of you using a Linux-based operating system, you're in luck - Linux is extremely developer-friendly and all of the tools you'll need are at your disposal from the command-line.

Unlike the Windows, when compiling code in Linux you won't need to set up a development environment using Visual Studio. Instead, you'll just set up a directory where you'll put and edit your C++ files, then will directly invoke the GNU C++ Compiler (g++) from the command-line.

If you're using Linux I'll assume that you're already familiar with simple commands like `mkdir` and `chdir` and that you know how to edit and save a text document. When writing C++ source code, you'll probably want to save header files with the `.h` extension and C++ files with the `.cpp`, `.C`, or `.c++` extension.

To be able to compile and run C and C++ programs, you will need first to install the build-essential package by typing the following command in the terminal (for Ubuntu distribution):

```
sudo apt-get install build-essential
```

This will install the necessary C/C++ development libraries for your Ubuntu Linux system to create C/C++ programs.

To compile your source code, you can execute g++ from the command line by typing g++ and then a list of the files you want to compile. For example, to compile myfile.cc and myotherfile.cc, you'd type:

```
g++ myfile.cc myotherfile.cc
```

By default, this produces a file named a.out, which you can execute by entering ./a.out. If you want to change the name of the program to something else, you can use g++'s -o switch, which produces an output file of a different name. For example, to create an executable called myprogram from the file myfile.cc, you could write

```
g++ myfile.cc -o myprogram
```

g++ has a whole host of other switches (such as -c to compile but not link a file), so be sure to consult the man pages for more info.

It can get tedious writing out the commands to compile every single file in a project to form a finished executable, so most Linux developers use makefiles, scripts which allow you to compile an entire project by typing the make command. A full tour of makefiles is far beyond the scope of an introductory C++ text, but fortunately there are many good online tutorials on how to construct a makefile. The full manual for make is available online at

<http://www.gnu.org/software/make/manual/make.html>.

## References

- [1] Brian W. Kernighan , and Dennis M. Ritchie, “*C Programming Language*”, 2<sup>nd</sup> Edition, 1988.
- [2] K. N. King , “*C Programming: A Modern Approach*”, 1996.
- [3] Dan Gookin, “*C For Dummies*”, 1997.
- [4] Online C course, Steve Summit:  
<http://www.eskimo.com/~scs/cclass/cclass.html>
- [5] Online C/C++ course, Paul Roberts, Cambridge University:  
[http://www-control.eng.cam.ac.uk/~pcr20/C\\_Manual/booktoc.html](http://www-control.eng.cam.ac.uk/~pcr20/C_Manual/booktoc.html)
- [6] Online C Resources:  
<http://www.tutorialspoint.com/cprogramming/>
- [7] MIT Open Courseware: Introduction to C++:  
<http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-096-introduction-to-c-january-iap-2011/>
- [8] C Lab Worksheets  
<http://www.tenouk.com/clabworksheet/clabworksheet.html>