C Language Basics

Types of Programming Languages

Different types of programming languages have evolved for humans to give *instructions* to *computers*:

1. **Machine language** – uses 0's and 1's to talk to the CPU/processor, but is *hard to learn* for programmers.

   e.g. ***0011 1100*** is an instruction that can be sent to an 8-bit processor as an instruction.

2. **Assembly language** – uses codes instead of 0's and 1's. *Easier* but still hard to learn.

   e.g. ***MOV A, B*** means move the contents of register A to register B.

3. **High Level Languages (HLLs)** – closer to *human language*. So, they are easier to learn and implement.

   Machine independent i.e. portable. Syntax and standards are well defined.

Types of Code for Programming in a HLL

The *Source Code* contains instructions written in the HLL. e.g. ***c= a+b;*** which means add the contents of variables ***a*** and ***b*** and store result in variable ***c***. Microprocessors don't understand instructions written in a HLL so the processor uses a *compiler* to *translate* the source code into *Object Code*. Object code consists of 0's and 1's. An object file is generated at this stage. e.g. ***0100 1000 1001 1100*** could be an instruction for a 16-bit processor.

HLL Program Execution Steps

To create an *executable* file from C source code, we first create a C program using an editor. There are then *four phases* required for a C program to become an executable file:

1. Pre-processing
2. Compilation
3. Assembly
4. Linking

The *pre-processing* phase includes:

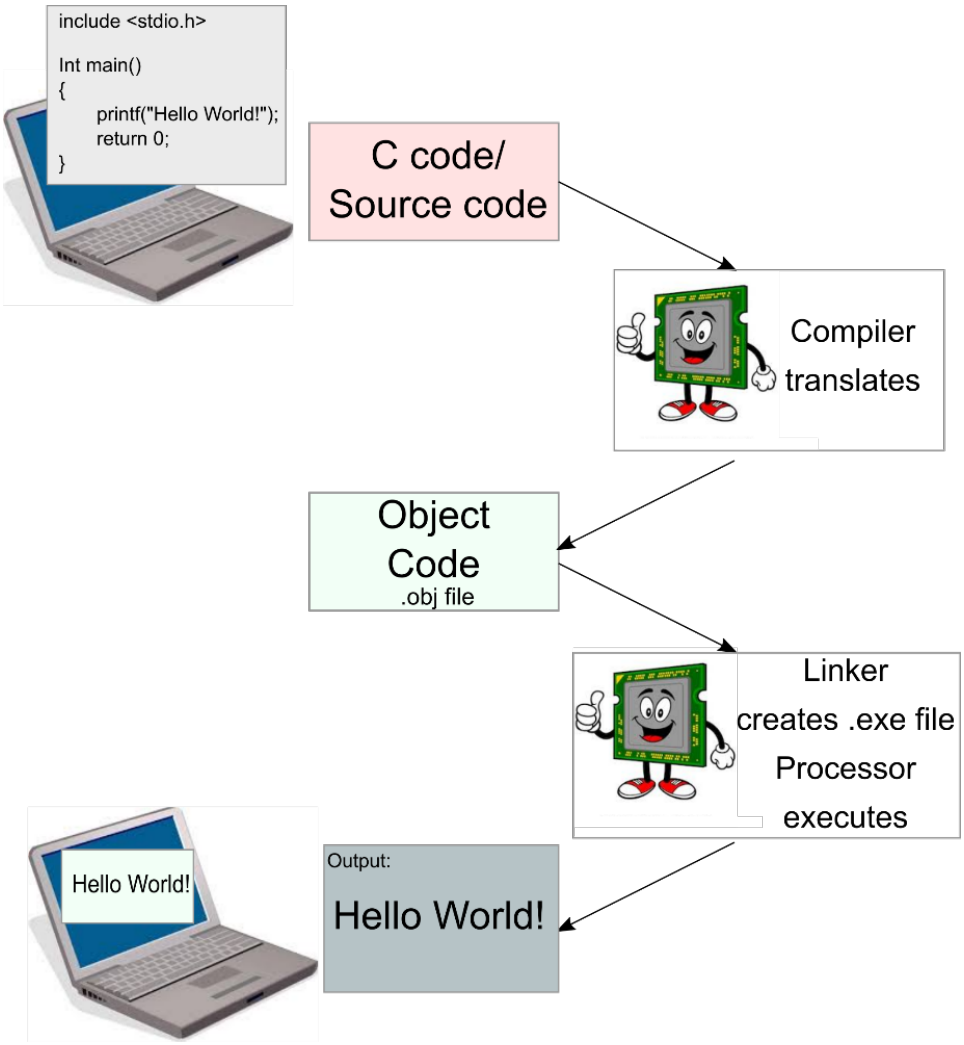
1. Removal of Comments
2. Expansion of Macros

3. Expansion of the included files

The next step is to *compile* the pre-processed code to produce assembly level instructions.

In the *assembly* phase, the assembly level instructions are translated to machine level instructions in the object code. At this phase, only code written by the programmer is converted into machine language, C function calls like **`printf`**() are not translated.

The final *linking* phase is where pre-defined C function calls like **`printf`**() are linked with their machine level definitions. The linker knows where all these definitions are stored.



Basic Elements of a C Program

The C language was created for programming the *UNIX operating system*. It was created by *Dennis Richie* in the 1970s and was derived from a language called *B*. C is often referred to as a *mid-level language* as it has the simplicity of a high-level language syntax and the *power* of a low-level language. As a result, programs written in C can be *very fast*.

Every full C program begins inside a *function* called “main”. The `main` function is *always called* when the program first executes. Other functions can be called from inside `main`. The `#include` statement is a *pre-processor directive* that tells the compiler to put code from the header file called “stdio.h” into your program *before* actually creating the executable. This effectively takes everything in the header file and *pastes* it into your program. By including header files, you can gain access to many different functions. For example, the **`printf`** function is included in “stdio.h”.

The statement **`int`** `main()` tells the compiler that there is a function named “main”, and that the function *returns* an integer. The *curly braces* `{` and `}` signal the beginning and end of functions and other *code blocks*. A code block is just a segment of code surrounded by curly braces. The **`printf`** function is the standard way in C for *displaying output* on the screen.

A *single line comment* starts with `//` and goes on until the end of the line. A *multi-Line comment* starts with `/*` and ends with `*/`. Everything in between `/*` and `*/` will be *ignored* by the compiler.

At the end of the program, a value is returned from `main` to the operating system by using the **`return`** statement. This return value is important as it can be used to tell the operating system whether the program *succeeded or not*. Usually, a return value of 0 means success.

```c
/*
This code illustrates the basic elements of a C program.
*/

#include <stdio.h>   // include the stdio.h library

// The main function starts here:

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

Expected Output:

```
Hello World!
```

Exercise 1

Run the following code and verify that it produces the expected output shown below.

```c
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

Expected Output:

```
Hello World!
```

Keywords

Keywords are reserved words that the C language uses for denoting something specific. The keywords are always written in *lowercase*. This table shows the 32 Keywords used in C:

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

Identifiers

Identifiers are the words we use to name entities like variables, functions, arrays, structures, symbolic constants etc.

The rules for naming identifiers are as follows:

1. Identifiers must consist of alphanumeric characters or underscores only.
2. The first character should be an alphabetic character or an underscore.
3. The identifier should not be a keyword.
4. Identifiers can be of any length.

Data Types

C uses different *data types* when storing data in memory. Each different type has a *specific range* and is stored using a *set number of bits*.

The four basic C data types are:

`char`: alphanumeric characters and special symbols e.g. ‘a’, ‘A’, ‘1’, ‘/’, …

`int`: integers (whole numbers)

`float`: floating point representation of real numbers

`double`: double precision real numbers

The C language also has the following *type qualifiers* which can be applied to the basic data types to define more types:

size qualifiers: **`short`**, **`long`**

sign qualifiers: **`signed`**, **`unsigned`**

When the **`unsigned`** qualifier is used, the number is always *positive*, and when **`signed`** is used the number may be positive or negative. If the qualifier is *not present* then the **`signed`** qualifier is assumed.

Typical Range and Stored Size of Integer Data Types on a 64-bit Computer

In C, the size of the data type is *machine dependent*. For an old 16-bit machine, the size of an **`int`** is 2 bytes. If, you are working on a 32-bit or 64-bit machine, then the size of an **`int`** is 4 bytes.

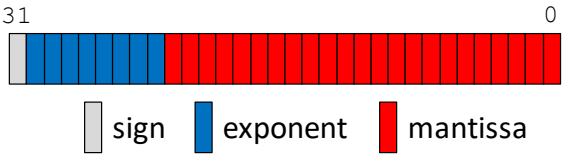
These tables show the typical *range* and *size* of different data types on a 64-bit machine:

Keyword	Range	Stored Size
`char`	-2^7 to $2^7 - 1$	1 byte
`short int`	-2^{15} to $2^{15} - 1$	2 bytes
`int`	-2^{31} to $2^{31} - 1$	4 bytes
`long int`	-2^{31} to $2^{31} - 1$	4 bytes
`long long int`	-2^{63} to $2^{63} - 1$	8 bytes

Keyword	Range	Stored Size
`unsigned char`	0 to $2^8 - 1$	1 byte
`unsigned short int`	0 to $2^{16} - 1$	2 bytes
`unsigned int`	0 to $2^{32} - 1$	4 bytes
`unsigned long int`	0 to $2^{32} - 1$	4 bytes
`unsigned long long int`	0 to $2^{64} - 1$	8 bytes

Bit Interpretation and Stored Size of Float Data Types

Floating point representations *vary* from machine to machine. Fortunately, one is by far the most common: the *IEEE-754 standard*. An IEEE-754 *float* (which is stored using 4 bytes) or *double* (which is stored using 8 bytes) has three components: a *sign bit* indicating whether the number is positive or negative, an *exponent* giving its order of magnitude, and a *mantissa* specifying the actual digits of the number. Using *single-precision* floats as an example, the bit layout of the 4 bytes looks like this:



The *value* of the floating point number is the *mantissa* multiplied by 2^E (2 to the power of the *exponent*). Notice that we are dealing with *binary fractions*, so that 0.1 (the most significant mantissa bit) means 1/2. The place values to the right of the decimal point are 2^{-1} (1/2), 2^{-2} (1/4), 2^{-3} (1/8) and so on. Just like we have 10^{-1} (1/10), 10^{-2} (1/100), and so on, in decimal. To represent *very small numbers* the value of the exponent must be *negative* so 127 and 1023 is subtracted from the binary exponent value for floats and doubles respectively. This table shows the *value*, *bit interpretation* and *size* of IEEE-754 floats and doubles:

Keyword	Value	Bit Interpretation	Size
`float`	$-1^S \times (1+M) \times 2^{(E-127)}$	$S = b_{31}$ $E = b_{30}2^7 + b_{29}2^6 + \ldots + b_{23}2^0$ $M = b_{22}2^{-1} + b_{21}2^{-2} + \ldots + b_0 2^{-23}$	$S = 1$ bit $E = 8$ bits $M = 23$ bits
`double`	$-1^S \times (1+M) \times 2^{(E-1023)}$	$S = b_{63}$ $E = b_{62}2^{10} + b_{61}2^9 + \ldots + b_{52}2^0$ $M = b_{51}2^{-1} + b_{50}2^{-2} + \ldots + b_0 2^{-52}$	$S = 1$ bit $E = 11$ bits $M = 52$ bits

Examples

Question:

Which decimal number is stored in this IEEE-754 **`float`**:

01000000011000000000000000000000

Solution:

	sign	exponent	mantissa
binary	0	10000000	11000000000000000000000
decimal	$S = 0$	$E = 128$	$M = 0.5+0.25 = 0.75$
formula terms	$-1^S = 1$	$E-127 = 1$	$1+M = 1.75$

solution 1 x 2^1 x 1.75 = 3.15

Question:

Which decimal number is stored in this IEEE-754 `float`:

10100000001010000000000000000000

Solution:

	sign	**exponent**	**mantissa**
binary	1	01000000	01010000000000000000000
decimal	S = 1	E = 64	M = 0.25+0.0625 = 0.3125
formula terms	-1^S = -1	E-127 = -63	1+M = 1.3125

solution -1 x 2^{-63} x 1.3125 = -1.42×10^{-19}

Exercise 2

Run the following code and verify that it produces the expected output shown below.

```c
#include<stdio.h>
#include<limits.h>
#include<float.h>

void main()
{
    printf("%22s %4s %24s\n", "", "Size", "Range");
    printf("%-22s %2lu %22d - %d\n", "char", sizeof(char), CHAR_MIN, CHAR_MAX);
    printf("%-22s %2lu %22hd - %hd\n", "short int", sizeof(short int), SHRT_MIN, SHRT_MAX);
    printf("%-22s %2lu %22d - %d\n", "int", sizeof(int), INT_MIN, INT_MAX);
    printf("%-22s %2lu %22ld - %ld\n", "long int", sizeof(long int), LONG_MIN, LONG_MAX);
    printf("%-22s %2lu %22lld - %lld\n\n", "long long int", sizeof(long long int), LLONG_MIN, LLONG_MAX);
    printf("%-22s %2u %22d - %d\n", "unsigned char", sizeof(unsigned char), 0, UCHAR_MAX);
    printf("%-22s %2u %22d - %d\n", "unsigned short int", sizeof(unsigned short int), 0, USHRT_MAX);
    printf("%-22s %2u %22d - %u\n", "unsigned int", sizeof(unsigned int), 0, UINT_MAX);
    printf("%-22s %2u %22d - %lu\n", "unsigned long int", sizeof(unsigned long int), 0, ULONG_MAX);
    printf("%-22s %2u %22d - %llu\n\n", "unsigned long long int", sizeof(unsigned long long), 0, ULLONG_MAX);
    printf("%-22s %2u %22le - %le\n", "float", sizeof(float), FLT_MIN, FLT_MAX);
    printf("%-22s %2u %22le - %le\n\n", "double", sizeof(double), DBL_MIN, DBL_MAX);
    printf("%-24s %4s\n", "", "Precision");
    printf("%-22s %1e\n", "float", FLT_EPSILON);
    printf("%-22s %1e\n\n", "double", DBL_EPSILON);
}
```

Expected Output

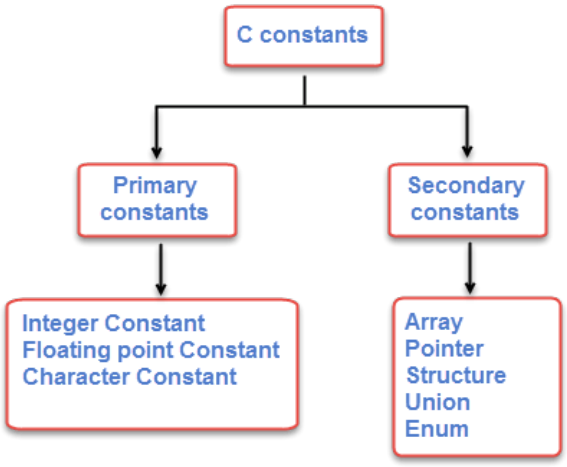
```
                       Size                    Range
char                    1                     -128 - 127
short int               2                   -32768 - 32767
int                     4              -2147483648 - 2147483647
long int                4              -2147483648 - 2147483647
long long int           8   -9223372036854775808 - 9223372036854775807

unsigned char           1                        0 - 255
unsigned short int      2                        0 - 65535
unsigned int            4                        0 - 4294967295
unsigned long int       4                        0 - 4294967295
unsigned long long int  8                        0 - 18446744073709551615

float                   4          1.175494e-038 - 3.402823e+038
double                  8          2.225074e-308 - 1.797693e+308

                         Precision
float                   1.192093e-007
double                  2.220446e-016
```

Constants

A *constant* (which is sometimes called a *literal*) is an entity that doesn't change. In C there are two types of constants:

1. *Primary* constants.
2. *Secondary* constants.

Both types are further divided into more categories as shown in this block diagram:



We will only look at *primary constants* for now. Some of the *secondary constants* will be discussed later.

Numeric Constants

Numeric constants are digits that may or may not have a decimal point. The rules for creating numeric constants are:

1. A constant must contain at least one digit.
2. No spaces, commas or any other special symbols are allowed.
3. A constant can be positive or negative. If no sign precedes a constant then it is assumed to be positive.

There are two types of numeric constants:

1. *Integer* constants
2. *Floating point* or *real* constants

Integer Constants

Integer constants do not have a decimal point. They can be written using *decimal* numbers (base 10), *octal* numbers (base 8) and *hexadecimal* numbers (base 16).

Decimal constants contain digits between **0** and **9**, but should not begin with **0**. For example:

```
43 199 3452 -100
```

Octal constants contain digits between **0** and **7**, and must begin with **0**. For example:

```
012 034 01144
```

Hexadecimal constants contain digits from **0** to **9**, and letters from **a**–**f** (either in uppercase or lowercase), and must always start with **0x** or **0X**. For example:

```
0x23 0Xff 0x37a 0XFF 0x37A
```

Floating Point or Real Constants – Fractional Form

Numeric constants which have a decimal point are called *floating point* or *real* constants.

Floating-point constants can be written in two forms:

1. Fractional form
2. Exponential form or Scientific notation

The rules for creating floating point constants in *fractional form* are:

1. Must have one at least one digit
2. Must have a decimal point
3. Can be positive or negative, the default is positive
4. No comma, blanks, or any other symbols are allowed

Here are some examples of floating point constants in fractional form:

```
3.14 899.0 -0.999
```

Floating Point or Real Constants – Exponential Form

Exponential form is used in cases when a number is too small or too large. For example, **0.00000941** can be represented as **9.41e-6**. The rules for creating floating point constants in exponential form are:

1. The mantissa and exponent must be separated by **`e`** or **`E`**.
2. The mantissa can be positive or negative, the default is positive.
3. The exponent must have at least one digit.
4. The exponent can be positive or negative, the default is positive

Here are some examples of floating point constants in exponential form:

```
00.34e4 -56E10 0.233E10 -0.94e15
```

ASCII

The American Standard Code for Information Interchange (which is usually referred to as *ASCII*) is a standard that defines how to translate from an *8-bit binary number* to an *alphanumeric character or symbol*. This table shows the ASCII translations:

Character	Hex	Decimal	Character	Hex	Decimal	Character	Hex	Decimal	Character	Hex	Decimal
NUL (null)	0	0	Space	20	32	@	40	64	.	60	96
Start Heading	1	1	!	21	33	A	41	65	a	61	97
Start Text	2	2	“	22	34	B	42	66	b	62	98
End Text	3	3	#	23	35	C	43	67	c	63	99
End Transmit.	4	4	$	24	36	D	44	68	d	64	100
Enquiry	5	5	%	25	37	E	45	69	e	65	101
Acknowlege	6	6	&	26	38	F	46	70	f	66	102
Bell	7	7	`	27	39	G	47	71	g	67	103
Backspace	8	8	(	28	40	H	48	72	h	68	104
Horiz. Tab	9	9	)	29	41	I	49	73	i	69	105
Line Feed	A	10	*	2A	42	J	4A	74	j	6A	106
Vert. Tab	B	11	+	2B	43	K	4B	75	k	6B	107
Form Feed	C	12	,	2C	44	L	4C	76	l	6C	108
Carriage Return	D	13	-	2D	45	M	4D	77	m	6D	109
Shift Out	E	14	.	2E	46	N	4E	78	n	6E	110
Shift In	F	15	/	2F	47	O	4F	79	o	6F	111
Data Link Esc	10	16	0	30	48	P	50	80	p	70	112
Direct Control 1	11	17	1	31	49	Q	51	81	q	71	113
Direct Control 2	12	18	2	32	50	R	52	82	r	72	114
Direct Control 3	13	19	3	33	51	S	53	83	s	73	115
Direct Control 4	14	20	4	34	52	T	54	84	t	74	116
Negative ACK	15	21	5	35	53	U	55	85	u	75	117
Synch Idle	16	22	6	36	54	V	56	86	v	76	118
End Trans Block	17	23	7	37	55	W	57	87	w	77	119
Cancel	18	24	8	38	56	X	58	88	x	78	120
End of Medium	19	25	9	39	57	Y	59	89	y	79	121
Substitue	1A	26	:	3A	58	Z	5A	90	z	7A	122
Escape	1B	27	;	3B	59	[	5B	91	{	7B	123

Character	Hex	Decimal	Character	Hex	Decimal	Character	Hex	Decimal	Character	Hex	Decimal
Form Separator	1C	28	<	3C	60	\	5C	92	|	7C	124
Group Separator	1D	29	=	3D	61	]	5D	93	}	7D	125
Record Separator	1E	30	>	3E	62	^	5E	94	~	7E	126
Unit Separator	1F	31	?	3F	63	_	5F	95	Delete	7F	127

Character Constants

A *character constant* is a *single* alphanumeric character or special symbol enclosed in *single* quotes. The maximum length of a character constant is *1 character long*. You *cannot* put *more than one* character inside the single quotation marks. Here are some examples of character constants:

```
'A' 'c' '4' '$' '^'
```

Consider this statement:

```
char ch = 'a'; // declare a variable ch and assign 'a' to it
```

Here we are declaring a variable `ch` of type **`char`** and assigning a character constant `'a'` to it. Although it might appear that we are assigning `'a'` to the variable `ch`, we are actually assigning the *ASCII value* of `'a'` (which has a decimal value of 97) to the variable `ch`.

String Constants

String constants consist of zero or more characters enclosed in *double quotes*. The *null* character (which has an ASCII value of 0) is *automatically* placed at the end of a string by the compiler. Here are some examples of string constants:

```
"hello" "123" ""
```

The *empty* string `""` consists of only the null character which is added by the compiler.

Although not *officially* a primary constant, we mention string constants here for completeness. C has no *string* data type as strings are stored as an *array of characters*.

Symbolic Constants

If we want to use a constant *several times* in a program, we can give it a name. For example, if there is a need to use the constant π = 3.141592 at several places in the program, we can give it a name and use *that name* instead of writing this long number. This constant is called a *symbolic constant* and is generally defined at the *beginning* of the program using a statement with this syntax:

```
#define NAME VALUE
```

`#define` is a pre-processor directive just like `#include`. `NAME` indicates the name we want to give to the constant and is generally written in *uppercase*. `VALUE` can be a numeric, character or string constant.

Let's create a symbolic constant called `PI` using this statement:

```
#define PI 3.14159
```

When the program is compiled, the pre-processor *replaces every occurrence* of `PI` by its value. For example, this statement:

```
printf("Circumference of a circle = %f", 2*PI*4);
```

will be compiled as if the statement was:

```
printf("Circumference of a circle = %f", 2*3.14159*4);
```

Use of symbolic constants makes a program more *maintainable* and *readable*. For example, let's say we wanted more accurate results, so we decided to update the value of π from 3.14159 to 3.14159265359. If we had *not used* a symbolic constant, we would have to go through the code and find *each occurrence* of **`3.14159`** and *update all of them*. However, since we have already defined `PI` in a `#define` directive, we only need to make the change in *one place*.

Variable Declarations

Variables are used to store data. As their name suggests, the value assigned to a variable can *change*. However, once you declare a variable to be a certain data type, you *can't change* the *type* of the variable later in the program.

Before you can use a variable you need to *declare* it. Declaring a variable involves specifying the *type* and *name* of the variable. Some example variable declarations are shown here:

```
int i;
char letter;
float x;
double d1;
```

you can declare *multiple variables* of the same type like this:

```
int a, b, c, d;
```

When a variable is declared it contains an *undefined value*. You can assign an *initial* value to the variable using the assignment operator (=). Assigning an initial value to the variable is called *initialization* of the variable. Here are some examples of variable initialization:

```
int a = 12, b = 100;
float f = 1.2;
char letter = 'a';
double d1, d2, d3 = 1.2;
```

In the last statement, *only* `d3` *is initialized*, `d1` and `d2` still contain an unknown value.

Exercise 3

Complete the following code so that it produces the expected output shown below.

```c
#include <stdio.h>

// Define a constant with the name PI and value 3.141592.

int main()
{
    // Declare an integer variable with the name radius.

    // Assign a value of 5 to the variable radius.

    printf("Circumference of the circle = %f\n", 2*PI*radius);

    return 0;
}
```

Expected Output

```
Circumference of the circle = 31.415920
```

Input and Output

Format Specifications

In this section we will discuss the input function **`scanf`**`()` and the output function **`printf`**`()`. Both of these functions require *format specifications* to describe the format of the input and output.

Each format specification begins with a `%` symbol. This table shows some common format specifications:

Format Specification	Description
`%c`	a single character
`%d`	an integer
`%f`	floating point number
`%x`	a hexadecimal integer
`%o`	an octal integer
`%i`	an integer, hexadecimal or octal
`%s`	a string
`%u`	an unsigned integer
`%h`	a short integer
`%l`	a long integer
`%ll`	a long long integer
`%le`	a long integer in exponential form

Printing Special Characters

This table shows format specifiers that print special characters when used as **`printf`**`()` format specifiers:

Format Specification	Description
`\a`	audible alert
`\f`	form feed
`\n`	newline, or linefeed
`\r`	carriage return
`\t`	tab
`\\`	print a backslash
`\'`	print a single quote
`\"`	print a double quote
`%%`	print a percent sign

Outputting Data

The **printf()** function is used to output data to the console. The syntax of this function is shown here:

```
printf("control string", variable1, variable2 , ...);
```

The *control string* controls how the output will appear on the screen and contains format specifications and text enclosed in *double quotes*.

The *variable arguments* specify which data to print on the console. These arguments can also be constants and expressions. The variable arguments are optional but each variable must have a format specification in the control string.

Example

```
#include<stdio.h>

int main()
{
    int ival = 100;
    double real_num = 32.78;
    char ch = 'a';

    printf("integer value = %d\n",ival);
    printf("real value = %f\n",real_num);
    printf("character value = %c",ch);

    return 0;
}
```

Expected Output

```
integer value = 100
real value = 32.780000
character value = a
```

Reading Input from the Keyboard

The **scanf()** function is used to read input from the keyboard. The syntax of this function is shown here:

```
scanf("control string", address1, address2 , ...);
```

The *control string* contains one or more format specifications enclosed in double quotes. The number of format specifications depends on the number of variables we want to input.

The next parameter, `address1` is the *memory address* of the variable that will be used to store the input data. We will discuss addresses later when we talk about pointers. The **`scanf`**`()` function expects at least one address. The address of the variable is designated by preceding a variable name with the `&` symbol.

Example

```c
#include<stdio.h>

int main()
{
    int i;

    printf("Enter a number: ");

    // accept input from keyboard
    scanf("%d", &i);

    printf("You entered %d", i);

    return 0;
}
```

Expected Output

```
Enter a number: 10
You entered 10
```

Exercise 4

Complete the following code so that it produces the expected output shown below.

```c
#include <stdio.h>

int main()
{
    // Declare three integer variables with the names:
    // length, width, area

    printf("Input the length of the rectangle: ");
    scanf("%d",&length);

    // Allow the user to input the width of the rectangle.

    // Calculate the area of the rectangle and assign this
    // value to the variable area.

    printf("Area of the rectangle = %d\n", area);
    return 0;
}
```

Expected Output

```
Input the length of the rectangle: 10
Input the width of the rectangle: 20
Area of the rectangle = 200
```

Formatting Integer Output

The *width* of the integer printed on the console can be controlled using an extra number in the format specifier of integer data types. This example demonstrates the use of the width format specification. If the length of the variable is *more than* the width specified, the output is printed correctly. If the length of the variable is *less than* the width specified, the value is printed right-justified with leading spaces. If a *zero* is inserted before the width specifier, the value is printed with leading zeros. If a minus sign is inserted before the width specifier, the value is printed left-justified.

```c
#include<stdio.h>

int main()
{
    int i = 23, j = 75;

    printf("i=%2d j=%2d\n",i,j);
    printf("i=%1d j=%1d\n",i,j);
    printf("i=%4d j=%4d\n",i,j);
    printf("i=%04d j=%04d\n",i,j);
    printf("i=%-4d j=%-4d\n",i,j);

    return 0;
}
```

Expected Output

```
i=23 j=75
i=23 j=75
i=  23 j=  75
i=0023 j=0075
i=23   j=75
```

Formatting Floating Point Output

The *number of decimal places printed* can be controlled using a decimal point and an extra number in the format specifier of floating point data types. This example demonstrates the use of the specification for the number of decimal places printed. If the variable has *more decimal places* than the number specified, the value is *rounded* before the output is printed. If the variable has *less decimal places* than the number specified, the value is printed with trailing zeros.

```c
#include<stdio.h>

int main()
{
    double i = 23.5492, j = 75.6972;

    printf("i=%-8.2f j=%-8.2f\n",i,j);
    printf("i=%-8.4f j=%-8.4f\n",i,j);
    printf("i=%-8.5f j=%-8.5f\n",i,j);

    return 0;
}
```

Expected Output

```
i=23.55    j=75.70
i=23.5492  j=75.6972
i=23.54920 j=75.69720
```

Exercise 5

Complete the following code so that it produces the expected output shown below.

```c
#include <stdio.h>

int main()
{
    // Declare three floating point variables with the names:
    // length, width, area

    printf("Input the length of the rectangle: ");
    scanf("%f", &length);

    // Allow the user to input the width of the rectangle.

    // Calculate the area of the rectangle and assign this
    // value to the variable area.

    printf("Area of the rectangle = ", area);

    // Print the value of the variable area
    // with 2 decimal places.

    return 0;
}
```

Expected Output

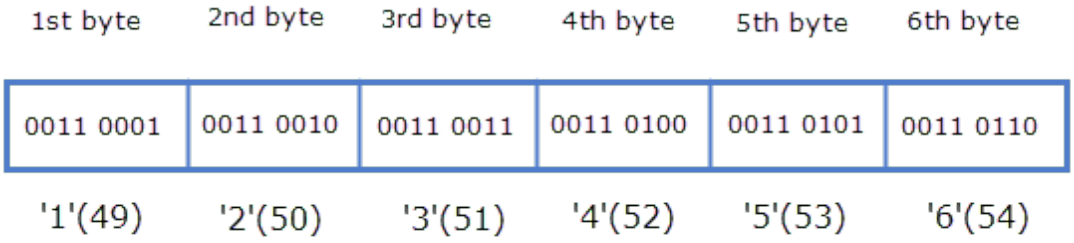
```
Input the length of the rectangle: 10.57
Input the width of the rectangle: 2.38
Area of the rectangle = 25.16
```

Storing Data in a File

In C we can store data in a file in two ways:

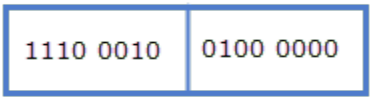
1. Text
2. Binary

In text mode, data is stored as a line of characters terminated by a newline character ('\n') where each character occupies 1 byte. To store 1234 in a text file would take 4 bytes, 1 byte for each character. The important thing to note is that in text mode what gets stored in the memory is the binary equivalent of the ASCII number of the character. Here is how 123456 is stored in the file in text mode:



As you can see from this example it takes 6 bytes to store 123456 in text mode.

In binary mode, data is stored on a disk in the same way as it is represented in computer memory. As a result, storing 123456 in a binary mode would take only 2 bytes. Here is how 123456 is stored in the file in binary mode:



Opening a File

Before any input/output can be performed with a file you must first open the file. This is the syntax of the **`fopen`**`()` function which is used to open a file:

```
FILE *fopen(const char *filename, const char *mode);
```

where `filename` is a string containing the name of the file and `mode` specifies what you want to do with the file.

On success the **`fopen`**`()` function returns a pointer to a structure of type `FILE`. (We will explain pointers and structures later.) The `FILE` structure is defined in stdio.h and contains information about the file like name, size, buffer size, current position, end of file etc.

On error the `fopen()` function returns `NULL`.

File Modes

Here are the possible values for `mode`:

1. `"w"` (write) – This mode is used to write data to the file. If the file doesn't exist this mode creates a new file. If the file already exists then this mode first clears the data inside the file before writing anything to it.
2. `"a"` (append) – This mode is called append mode. If the file doesn't exist this mode creates a new file. If the file already exists then this mode appends new data to end of the file.
3. `"r"` (read) – This mode opens the file for reading. To open a file in this mode file must already exist. This mode doesn't modify the contents of the file in anyway. Use this mode if you only want to read the contents of the file.
4. `"w+"` (write + read) – This mode is a same as `"w"` but in this mode, you can also read the data. If the file doesn't exist this mode creates a new file. If the file already exists then previous data is erased before writing new data.
5. `"r+"` (read + write) – This mode is same as `"r"` mode , but you can also modify the contents of the file. To open the file in this mode file must already exist. You can modify data in this mode but the previous contents of the file are not erased. This mode is also called update mode.
6. `"a+"` (append + read) – This mode is same as `"a"` mode but in this mode, you can also read data from the file. If the file doesn't exist then a new file is created. If the file already exists then the new data is appended to the end of the file. Note that in this mode you can append data but can't modify existing data.

To open the file in binary mode you need to append `"b"` to the mode like this:

Mode	Description
`"wb"`	Open the file in binary mode
`"a+b"` or `"ab+"`	Open the file in append + read in binary mode

The `fscanf` Function

The `fscanf()` function is used to read formatted input from a file. This is the syntax of the function:

```
int fscanf(FILE *fp, const char *format [, argument, ...] );
```

It works just like `scanf()` function but instead of reading data from the standard input it reads the data from a file. The arguments of `fscanf()` are the same as `scanf()`, except it needs the

additional argument `fp` which is a file pointer. On success, this function returns the number of values read and on error or if it reaches the end of the file it returns `EOF` or **`-1`**.

The `fprintf` Function

The **`fprintf`**`()` function is used to write formatted output to a file. This is the syntax of the function:

```
int fprintf(FILE *fp, const char *format [, argument, ...] );
```

It works just like **`printf`**`()` but instead of writing data to the console it writes data to a file. The arguments of **`fprintf`**`()` are the same as **`printf`**`()`, except it needs the additional argument `fp` which is a file pointer. On success, this function returns the total number of characters written to the file. On error, it returns `EOF`.

Example

```c
#include<stdio.h>

int main()
{
    FILE *fp;
    char *name, *name_in;
    int id, id_in, chars;
    float score, score_in;

    id = 7803017;
    name = "Pickering";
    score = 85.7;

    fp = fopen("student records.txt", "w+");

    if(fp == NULL)
    {
        printf("Error opening file\n");
        exit(1);
    }

    printf("Testing fprintf() function: \n\n");

    chars = fprintf(fp, "Name: %s\t\tID: %7d\t\tMark: %6.2f\n", name, id, score);

    printf("%d characters successfully written to the file:\n", chars);
    printf("Name: %s\t\tID: %7d\t\tMark: %6.2f\n\n", name, id, score);

    rewind(fp);

    printf("Testing fscanf() function: \n\n");
    fscanf(fp, "Name: %s\t\tID: %d\t\tMark: %f\n", name_in, &id_in, &score_in);

    printf("characters read from the file:\n");
    printf("Name: %s\t\tID: %7d\t\tMark: %6.2f\n", name_in, id_in, score_in);

    fclose(fp);
    return 0;
}
```

Expected Output

```
Testing fprintf() function:

43 characters successfully written to the file:
Name: Pickering         ID: 7803017             Mark:  85.70

Testing fscanf() function:

characters read from the file:
Name: Pickering         ID: 7803017             Mark:  85.70
```

Exercise 6

Complete the following code so that it produces the expected output shown below.

```c
#include<stdio.h>

int main()
{
    FILE *fp;
    int chars;
    float height, height_in;

    printf("Enter your height in centimeters: ");
    scanf("%f", &height);

    // Open the file "height.txt" in write + read mode.
    // Assign the file pointer fp to the file.

    // Check to see if the file was opened correctly.
    // If not, display an error message and exit the program.

    printf("\nTesting fprintf() function: \n\n");

    // Write the value of height to the file "height.txt".
    // The output should be formatted as:
    // Height: nnn.nn
    //
    // Assign the number of characters correctly written
    // to the variable chars.

    printf("%d characters were written to the file:\n", chars);
    printf("Height: %6.2f\n\n", height);

    rewind(fp);

    printf("Testing fscanf() function: \n\n");

    // Read a value of height from the file "height.txt".
    // The file is formatted as:
    // Height: nnn.nn
    //
    // Assign the value to the variable height_in.
```

```
    printf("characters read from the file\n");
    printf("Height: %6.2f\n\n", height_in);

    // Close the file "height.txt".

    return 0;
}
```

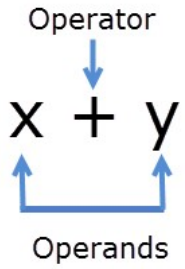
Expected Output

```
Enter your height in centimeters: 178

Testing fprintf() function:

15 characters successfully written to the file:
Height: 178.00

Testing fscanf() function:

characters read from the file:
Height: 178.00
```

Expressions and Operators in C

Operators and Operands

An *operator* specifies an operation on the data which yields a value. The data item on which the operator acts is called an *operand*.



Some operators need two operands while some need only one. C language provides the following operators:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Conditional Operators
5. Assignment Operators
6. Bitwise Operators
7. sizeof Operator

Arithmetic Operators

This table lists the arithmetic operators in C:

Operator	Description
+	plus
−	minus
*	multiply
/	divide
%	remainder

The first four operators work as usual, but you might not have seen the `%` operator. The `%` operator is known as the *remainder* or *modulus* operator. The remainder operator is used to calculate the remainder of a division operation. For example, **`9%2`** would produce **`1`**. An important thing to note is that the remainder operator only works with *integers*, you can't apply the `%` operator on **`float`** or **`double`** types.

Integer Arithmetic

When both operands are integers then the result of the arithmetic operation between two integer operands yields an *integer* value. Let's take two variables `a` and `b`, where `a = 10` and `b = 4`. This table shows the results of arithmetic operations performed on `a` and `b`.

Expression	Result
a+b	14
a-b	6
a*b	40
a/b	2
a%b	2

We know that **`10/4 = 2.5`**, but because both operands are *integers*, the decimal value is truncated. For the division and remainder operators to work, the second operand must be *non-zero*, otherwise, the program will crash.

Floating Point Arithmetic

An operation between two floating point operands always yields a *floating point* result. Let's take two variables `a` and `b`, where `a = 11.2` and `b = 4.5`. This table shows the results of arithmetic operations performed on `a` and `b`.

Expression	Result
a+b	15.700000
a-b	6.700000
a*b	50.399999
a/b	2.488889

Mixed Mode Arithmetic

An operation between an integer and a floating point yields a *floating point* result. In this operation, the integral value is first *converted to a floating point value* and then the operation is performed. Let's take two variables `a` and `b`, where `a = 14` and `b = 2.5`. This table shows the results of arithmetic operations performed on `a` and `b`.

Expression	Result
a+b	16.500000
a-b	11.500000
a*b	35.000000
a/b	5.600000

As we can see, when **14** is divided by **2.5** the fractional part is not lost because an arithmetic operation between an **int** and a **float** yields a **float** value. So we can use mixed mode arithmetic to solve the problem we encountered while dividing **10/4**. To get the correct answer simply make one of the operands involved in the operation a floating point number. For example, **10/4.0** or **10.0/4** both will give the correct result of **2.5**.

Relational Operators

Relational operators are used to compare values of two expressions. Relational operators are binary operators because they require two operands to operate. An expression which contains the relational operators is called a relational expression. If the relation is true then the result of the relational expression is **1**, if the relation is false then the result of the relational expression is **0**. This table shows the C relational operators:

Operator	Description
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than or equal to
==	is equal to
!=	is not equal to

Logical Operators

Logical operators are used to evaluate two or more conditions. In general, logical operators are used to combine relational expressions, but they are not limited to just relational expression. You can use logical operators with any kind of expression, even constants. In C, all *non-zero values* are considered to be a *logical true* while 0 is considered to be a *logical false*. If the result of the logical operator is true then **1** is returned otherwise **0** is returned. This table shows the C logical operators:

Operator	Description
`&&`	logical AND
`||`	logical OR
!	logical NOT

Conditional Operator

The conditional operator (`?` and `:`) is a special operator which requires three operands:

```
expression1 ? expression2 : expression3
```

If `expression1` is *true* then the result of the overall expression becomes `expression2`. If `expression1` is *false*, then the result of the overall expression becomes `expression3`.

For example, this code assigns the greater of two variable values to the variable `max` using the conditional operator:

```
max = a > b ? a : b;
```

If `a` is greater than `b` then the expression `a > b` is *true* and the statement is evaluated as `max = a`. If `b >= a` then the expression `a > b` is *false* and the statement is evaluated as `max = b`.

Assignment Operator

The assignment operator (=) is used to assign a value to a variable. The operand on the left-hand side of the assignment operator must be a variable and the operand on the right-hand side must be a constant, variable or expression. Here are some examples:

```
x = 18;     // right operand is a constant
y = x;     // right operand is a variable
z = 1 * 12 + x;     // right operand is an expression
```

Compound Assignment Operator

Consider the statement:

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

Here the right-hand side adds **5** to the existing value of `x`. This value is then assigned back to `x`. To handle operations like this more concisely, C provides a special operator called a *compound assignment* operator. The general format of the compound assignment operator is:

```
variable op= expression
```

where `op` can be any of the arithmetic operators (`+`, `-`, `*`, `/`, `%`).

This statement is equivalent to:

```
variable = variable op (expression)
```

For example, the statement `x += 5;` is equivalent to `x = x + 5;`.

Increment and Decrement Operators in C

C has two special unary operators called increment (`++`) and decrement (`--`) operators. These operators increment and decrement the value of a variable by **1**. For example:

`++x` is the same as `x = x + 1` or `x += 1`
`--x` is the same as `x = x - 1` or `x -= 1`

The Increment and decrement operators can only be used with variables. They can't be used with constants or expressions. There are two types of Increment/Decrement operators:

1. Prefix increment/decrement operator.
2. Postfix increment/decrement operator.

Prefix and Postfix Increment/Decrement Operator

The *prefix* increment/decrement operator immediately increases or decreases the current value of the variable. This value is then used in the expression. For example, in this statement:

```
y = ++x;
```

the current value of `x` is incremented by **1** and then the new value of `x` is assigned to `y`.

For the *postfix* increment/decrement operator, the current value of the variable is used in the expression. Then the value of the variable is increased or decreased.

For example, in this statement:

```
y = x++;
```

the current value of `x` is assigned to `y` and then the value of `x` is incremented by **1**.

Bitwise Operators

Bitwise operators are used for manipulating data at the bit level. This table shows the C bitwise operators:

Operator	Description
<<	shift left
>>	shift right
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR
~	ones complement

Example

This program shows the use of bitwise operators:

```c
#include <stdio.h>

int main()
{
    unsigned char a = 5, b = 9;

    printf("a = %s (%d)\nb = %s (%d)\n\n", "00000101", a, "00001001", b);

    printf("AND: a&b = %s (%d)\n", "00000001", a & b);

    printf(" OR: a|b = %s (%d)\n", "00001101", a | b);

    printf("XOR: a^b = %s (%d)\n", "00001100", a ^ b);

    printf("NOT:  ~a = %s (%d)\n\n", "11111010", a = ~a);

    printf(" Shift Left: b<<1 = %s (%d)\n", "00010010", b << 1);

    printf("Shift Right: b>>1 = %s (%d)\n", "00000100", b >> 1);

    return 0;
}
```

Expected Output

```
a = 00000101 (5)
b = 00001001 (9)

AND: a&b = 00000001 (1)
 OR: a|b = 00001101 (13)
XOR: a^b = 00001100 (12)
NOT:  ~a = 11111010 (250)

 Shift Left: b<<1 = 00010010 (18)
Shift Right: b>>1 = 00000100 (4)
```

Bitwise operators can be used in integer calculations to reduce the execution time of code. For example, `b<<1` executes much faster than `b*2`.

`sizeof` Operator

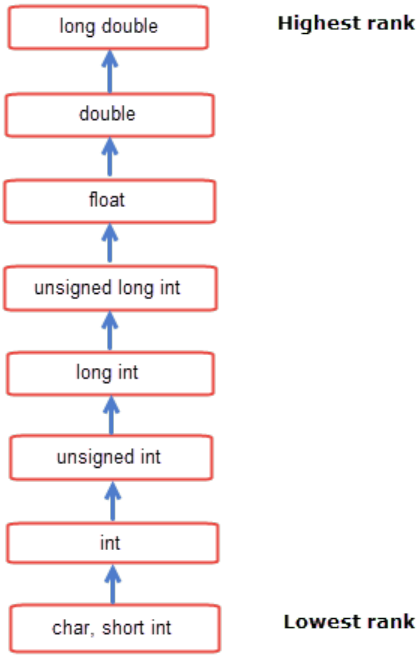
The unary operator `sizeof()` is used to determine the size of its operand. The syntax for the `sizeof()` operator is:

```
sizeof(object)
```

where `object` can be a data type keyword like `int`, `float`, `double` or an expression or a variable. For example, `sizeof(int)` gives the size occupied by an `int` data type. The `sizeof()` operator returns size in bytes.

Implicit Type Conversion in C

In C, data types have a *rank order* as shown in this diagram:



In expressions that contain two operands of *different types*, one operand is *automatically* converted to the other type by the *compiler*. This process is known as *implicit type conversion*. The operand with the *lower rank* will be converted to the data type of the operand with the *higher rank*.

Type Conversion in Assignment

If the types of the operands in an assignment expression are different, the operand on the *right-hand side* will be converted to the type of the *left-hand operand*. Some consequences of type conversion in an assignment expression are:

1. High order bits may be lost when `long int` is converted to `int` or `int` to `short int` or `char`.
2. The fractional part will be truncated during conversion from floating point types to integer types.
3. When a `double` is converted to a `float` digits are rounded off.
4. When an integer type is converted to a `float` or a `float` is converted to a `double` there is no increase in accuracy.
5. When a `signed` type is changed to an `unsigned` type, the sign may be dropped.

Explicit Type Conversion

The type of a constant, variable or expression can be temporarily converted to another type using the *cast operator*. The syntax of the cast operator is:

```
(datatype)expression
```

where `datatype` is the type you want the expression to be converted to.

Operator Precedence and Associativity in C

Operator precedence dictates the *order* in which operators in an expression will be evaluated. *Associativity* defines the order in which operators of the *same precedence* are evaluated in an expression. Associativity can be either from *left to right* or *right to left*. Operator precedence and associativity in C is shown in this table:

Operator	*Description*	*Associativity*
() **[]** **.** **->** **++ --**	Parentheses or function call Brackets or array subscript Dot or Member selection operator Arrow operator Postfix increment/decrement	left to right
++ -- **+ -** **! ~** **(type)** ***** **&** **sizeof**	Prefix increment/decrement Unary plus and minus not operator and bitwise complement type cast Indirection or dereference operator Address of operator Determine size in bytes	right to left

Operator	*Description*	*Associativity*
* / %	Multiplication, division and modulus	left to right
+ -	Addition and subtraction	left to right
<< >>	Bitwise left shift and right shift	left to right
< <= > >=	relational less than/less than equal to relational greater than/greater than or equal to	left to right
== !=	Relational equal to and not equal to	left to right
&	Bitwise AND	left to right
^	Bitwise exclusive OR	left to right
|	Bitwise inclusive OR	left to right
&&	Logical AND	left to right
||	Logical OR	left to right
? :	Ternary operator	right to left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment operator Addition/subtraction assignment Multiplication/division assignment Modulus and bitwise assignment Bitwise exclusive/inclusive OR assignment	right to left
,	Comma operator	left to right

Exercise 7

Complete the following code so that it produces the expected output shown below.

```c
#include <stdio.h>

int main()
{
    char y_char, x_char = 126;
    unsigned char y_uchar, x_uchar = 126;
    int y_int, x_int = 126;
    float y_float, x_float = 126;

    // Multiply the variables
    // x_char, x_uchar, x_int and x_float
    // by 4 and assign the answers to
    // y_char, y_uchar, y_int and y_float.

    printf("126x4\n\n");
    printf("char:   %d\n",y_char);
    printf("u char: %u\n",y_uchar);
    printf("int:    %d\n",y_int);
    printf("float:  %6.2f\n",y_float);

    // Divide the variables
    // x_char, x_uchar, x_int and x_float
    // by 4 and assign the answers to
    // y_char, y_uchar, y_int and y_float.

    printf("\n\n126/4\n\n");
    printf("char:   %d\n",y_char);
    printf("u char: %u\n",y_uchar);
    printf("int:    %d\n",y_int);
    printf("float:  %5.2f\n",y_float);

    return 0;
}
```

Expected Output

```
126x4

char:   -8
u char: 248
int:    504
float:  504.00


126/4

char:   31
u char: 31
int:    31
float:  31.50
```

Control Statements in C

Control Statements

Control statements are used to alter the *flow* of the program and are used to specify the *order* in which statements can be executed. They are commonly used to define how control is transferred from one part of the program to another.

The C language has these control statements:

1. if … else
2. while loop
3. do … while loop
4. for loop
5. switch

The `if` Statement

The `if` statement is used to test a condition and take one of the two possible actions. The syntax of the `if` statement is shown here:

```
if(condition)
{
    statement_1;
          .
          .
    statement_n;
}
```

where `condition` can be any constant, variable, expression, relational expression, logical expression and so on. Just remember that in C, any *non-zero value* is considered to be a logical true while `0` is considered as a logical false.

The statements inside the if block are executed only when the condition is true. If it is false then statements inside if the block are skipped. The braces (`{}`) are always required when you want to execute more than one statement when the condition is true. Also, note that the statements inside the `if` block are slightly indented. This is done to improve readability, indentation is not syntactically required.

If you only want to execute *one* statement when the condition is true then the braces can be omitted.

The `else` Statement

The **`else`** clause allows an *alternative path* to be added to the **`if`** condition. Statements in the **`else`** block are executed only when the **`if`** condition is *false*.

```
if(condition)
{
    statement1;
    statement2;
}
else
{
    statement3;
    statement4;
}
```

The `else if` Statement

The **`else if`** statement *extends* the basic **`if else`** statement and allows a *series of tests* to be performed.

```
if(condition1)
{
    statement1;
}
else if(condition2)
{
    statement2;
}
...
else
{
    statement3;
}
```

Each condition is checked *one by one*. As *soon as a condition is found to be true* then statements corresponding to *that block* are executed. The conditions and statements in the *rest* of the **`if else`** statement are *skipped* and program control comes out of the **`if else`** statement. If *none* of the conditions are true then the statements in the **`else`** block are executed.

The `while` Loop

Loops are used to execute statements or a block of statements *repeatedly*. The syntax of the while loop is shown here:

```
while(condition)
{
    statement1;
    statement2;
}
```

If `condition` is true, the statements in the while block are executed. After executing these statements, the condition is checked again, if it is *still true* then the statements in the **`while`** block are executed *again*. This process keeps repeating until `condition` becomes *false*. Therefore, you must always include a statement, *inside the* **`while`** *block*, which alters the value of the `condition` so that it *ultimately* becomes *false* at some point. Each execution of the loop is known as an *iteration*.

Example

```c
#include <stdio.h>

int main()
{
    int num, rem;

    num = 1;

    // keep looping while num is not equal to zero
    while( num != 0 )
    {
        printf("Enter a number (0 to quit): ");
        scanf("%d", &num);

       // if the number entered is not zero
       // print whether it is odd or even
        rem = num%2;
        if(num != 0)
        {
            if(rem)
            {
                printf("remainder = %d, %d is an odd number\n\n",rem,num);
            }
            else
            {
                printf("remainder = %d, %d is an even number\n\n",rem,num);
            }
```

```
        }
      }

    printf("Thanks for playing\n\n");
    return 0;
}
```

Expected Output

```
Enter a number (0 to quit): 2
remainder = 0, 2 is an even number

Enter a number (0 to quit): 5
remainder = 1, 5 is an odd number

Enter a number (0 to quit): -5
remainder = -1, -5 is an odd number

Enter a number (0 to quit): 0
Thanks for playing
```

The `do while` Loop

The syntax of a `do while` loop is shown here:

```
do{
    statement1;
    statement2;
}while(condition);
```

In a `do while` loop, the statements in the body are executed *first*, then the condition is checked. If the condition is *true* then the statements in the block are *executed again*. This process keeps repeating until the condition becomes false. Notice that, unlike the while loop, a do while statement needs a *semicolon* (*;*) after the condition.

The `do while` loop differs significantly from the `while` loop because in a `do while` loop the statements in the block are executed *at least once* even if the condition is *false*.

The `for` Loop

The syntax of the `for` loop is shown here:

```
for(expression1; expression2; expression3)
{
    statement1;
    statement2;
}
```

Where `expression1` is the initialization expression, `expression2` is the test expression or condition, and `expression3` is the update expression. First, the *initialization expression* (`expression1`) is executed to initialize loop variables. This expression is executed *only once* when the loop starts. Then the *condition* (`expression2`) is checked. If it is *true*, the body of the loop is executed. After executing the loop body, the program control is transferred to the *update expression* (`expression3`). This expression modifies the loop variables. Then the condition (`expression2`) is checked again. If the condition *is still true*, the body of the loop is executed once more. This process *continues* until `expression2` becomes *false*.

Example

This code uses a `for` loop to find the sum of the numbers between 1 and 100.

```
#include<stdio.h>

int main()
{
    int i;          // loop variable
    int sum = 0;    // variable to accumulate sum

    for(i = 1; i <= 100; i++)
    {
        sum += i;
    }

    printf("Sum = %d\n", sum);
    return 0;
}
```

Expected Output

```
Sum = 5050
```

`break` and `continue`

Two keywords that are very important to looping are `break` and `continue`. The `break` command will exit the most immediately surrounding loop regardless of what the conditions of the loop are. The `break` keyword is useful if we want to exit a loop under special circumstances.

`continue` is another keyword that controls the flow of loops. If you are executing a loop and hit a `continue` statement, the loop will stop its current iteration, update itself (in the case of `for` loops) and begin to execute again from the top. Essentially, the `continue` statement is saying "this iteration of the loop is done, let's continue with the loop without executing whatever code comes after me."

The `switch` Statement

The `switch` statement is a *multi-directional* statement used to handle *decisions*. It works almost exactly like the `if else` statement. The *difference* is that the `switch` statement produces a more *readable* code. The syntax of a switch statement is shown here:

```
switch(expression)
{
    case constant1:
        statement1;
        ...
    case constant2:
        statement2;
        ...
    default:
        statement3;
        ...
}
```

The expression in the `switch` statement can be any valid expression which yields an *integer value*. It can also be a *character constant* but it *can't be* a floating point or a string. The constants following the `case` keywords must be of *type integer or character*. It can also be an expression which *yields an integer value*. Each `case` statement must have *only one constant* and all case constants must be *unique*. After each case constant, you can have *any number* of statements or *no statement at all*. If there are multiple statements, you *don't* need to enclose them with braces ({ }).

First, the expression following the `switch` is evaluated, then the value of this expression is compared against *every* `case` one by one. If the value of the expression matches with *any* `case` *constant* then

the statements under that **case** are executed. If the value of the expression *does not match* any **case** constants then the statements under the **default** keyword are executed. The **default** statement is *optional*. If it is omitted and no case matches then *no action* takes place.

Example

Notice the use of the **break** statements in the following example. If we left the **break** statements out, when a match is found, *all statements* under that **case** statement are executed. The important thing to note is that statements under *any following* **case** *statements* and the **default** statement *will also be executed*. This is known as *falling through cases* and this is how the **switch** statement works *by default*. **break** statements are required *under all cases* if you *don't want* a **switch** statement to fall through cases.

```c
#include<stdio.h>

int main()
{
    int i, sum;

    printf("Enter a number: ");
    scanf("%d", &i);

    switch(i)
    {
        case 1:
            printf("Number is one\n");
            break;
        case 2:
            printf("Number is two\n");
            break;
        default:
            printf("something else\n");
    }

    return 0;
}
```

Expected Output

First run:

```
Enter a number: 3
Number is three
```

Second run:

```
Enter a number: 11
something else
```

Exercise 8

Complete the following code so that it produces the expected output shown below.

```c
#include<stdio.h>

int main()
{
    int i;          // loop variable
    int sum = 0;    // variable to accumulate sum
    int mean = 0; // variable to store mean value

    // Use a for loop to calculate the
    // mean of the numbers between
    // 50 and 150 inclusive.

    printf("mean = %d\n",mean);

    return 0;
}
```

Expected Output

```
mean = 100
```

Functions

What is a Function?

A function is a collection of C statements to do something specific. A C program consists of one or more functions. Every program must have a function called `main()`.

The advantages of using functions include:

1. A large problem can be divided into *subproblems* and then solved by using functions.
2. The functions are *reusable*. Once you have created a function you can call it *anywhere* in the program without copying and pasting *entire bocks* of code.
3. The program becomes more *maintainable*. If you want to modify the program sometime later, you only need to update your code in *one place*.

You can either use the built-in C library functions or you can *create your own* functions. To create your own function you need to know about *three things*:

1. the function definition
2. the function call
3. the function declaration

The Function Definition

The function *definition* is the the *actual code* of the function. A function consists of two parts: the function *header* and the function *body*. Here is the general syntax of a function:

```
return_type function_name(type1 argument1, type2 argument2, ...)
{
    local variables;

    statement1;
    statement2;

    return(expression);
}
```

The first line of the function is known as the function header. It consists of `return_type`, `function_name` and function *arguments*. `return_type` indicates the *type* of the value that the function returns e.g. **`int`**, **`float`** etc. `return_type` is *optional*, if it is omitted then the return type

is assumed to be **int** by default. A function can either return *one value* or *no value at all*, if a function doesn't return any value, then the keyword **void** is used in place of `return_type`.

`function_name` is the name of the function. It can be any valid C identifier. After the name of the function, we have the *arguments declaration* inside parentheses. Each declaration consists of the *type* and *name* of the argument. A function can have *any number* of arguments or even *no arguments* at all. If the function has *no arguments* then the parentheses are *left empty* or sometimes the keyword **void** is used to represent a function which accepts no arguments.

The *body* of the function is a *block of statements* which consists of any valid C statements followed by an optional **return** statement. The variables declared *inside the function* are called *local* variables because they are local to the function. This means *you can't access* the variables declared *inside one function* from *another function*. The **return** statement is used when a function needs to return something to its caller. The **return** statement is optional. If a function doesn't return any value then it's `return_type` must be **void**, similarly if a function returns an **int** value its `return_type` must be **int**.

You can write function definitions anywhere in the program, but usually, they are placed after the `main()` function.

Example

Here is an example function definition:

```
int product(int num1, int num2)
{
    int result;

    result = num1 * num2;

    return(result);
}
```

This function accepts *two arguments* and returns an *integer* value. The variable `result` is declared inside a function, so it is a local variable and *only available inside the function*. The **return** statement returns the product of `num1` and `num2` to its caller. Another important point to note is that `num1` and `num2` are *also local variables*, which means we can't access them *outside the function* `product()`.

The Function Call

After the function is defined the next step is to *use the function*. To use the function you must *call it*. To call a function you must write its name followed by arguments, separated by a comma, inside the parentheses. For example, here is how we can call the `product()` function we just created:

```
product(12,10);
```

Here we are passing two arguments, **12** and **10**, to the function `product()`. The values **12** and **10** will be assigned to variables `num1` and `num2` respectively.

If a function returns a value then it can be used *inside any expression* like an operand. These statements show examples of using a function call as an expression:

```
a = product(34,89) + 100;
printf("product is = %d",product(a,b));
```

The Function Declaration

The *calling function* needs some information about the *called function*. Generally function definitions come *after* the `main()` function. In this case, a *function declaration* is needed. A function declaration consists of the function header with a semicolon (**;**) at the end. Here is the function declaration for the function `product()`:

```
int product(int x, int y);
```

The names of arguments in a function declaration are *optional* but the return type and argument types must be *the same* as in the function definition. When the function definition comes *before* the calling function, a function declaration is *not* needed.

Local, Global and Static Variables

This table describes the different properties of local, static and global variables:

Type	Declared	Scope	Retains Value
local	inside function	inside declaring function	no
static	inside function	inside declaring function	yes
global	not inside any function	everywhere	yes

Variables which are declared inside a function are called *local* variables. They are only available inside the function in which they are declared. A *static* variable is able to *retain its value* between different function calls. A static variable is only *initialized once*. If it is not initialized, it is automatically initialized to **0**. Variables declared outside any function are called *global* variables. They are *not limited* to any function. Any function can *access and modify* global variables. Global variables are automatically initialized to **0** at the time of declaration. Global variables are generally written before the `main()` function.

Example

This example shows how to declare local, global and static variables:

```c
#include<stdio.h>

void func_1();  // function declaration for func_1

int a = 1;  // declaring and initializing a global variable

int main()
{
    printf("from inside main() global a = %d\n\n", a++);
    func_1(); // function call for func_1
    printf("from inside main() global a = %d\n\n", a++);
    func_1();// function call for func_1
    printf("from inside main() global a = %d\n\n", a++);
    func_1();// function call for func_1

    return 0;
}

// function definition for func_1
void func_1()
{
    int b = 10;  // declaring and initializing a local variable
    static int c = 100;  // declaring and initializing a static variable

    printf("from inside func_1() global a = %d\n", a++);
    printf("from inside func_1() local b = %d\n", b++);
    printf("from inside func_1() static c = %d\n\n", c++);
}
```

Expected Output

```
from inside main() global a = 1

from inside func_1() global a = 2
from inside func_1() local b = 10
from inside func_1() static c = 100

from inside main() global a = 3

from inside func_1() global a = 4
from inside func_1() local b = 10
from inside func_1() static c = 101

from inside main() global a = 5

from inside func_1() global a = 6
from inside func_1() local b = 10
from inside func_1() static c = 102
```

Exercise 9

Complete the following code so that it produces the expected output shown below.

```c
#include<stdio.h>

// Declare a function called my_sum
// to find the sum of two integers.
// It should accept 2 integer arguments
// and return an integer answer.

int main()
{
    int x, a, b;

    printf("Enter the value for a: ");
    scanf("%d", &a);
    printf("Enter the value for b: ");
    scanf("%d", &b);

    // Call the function my_sum
    // to add a and b and assign the
    // answer to x.

    printf("\n\nThe sum of a and b = %d \n\n",x);

    return 0;
}

// Define the function my_sum.
```

Expected Output

```
Enter the value for a: 10
Enter the value for b: 12


The sum of a and b = 22
```

Pointers

What is a Pointer?

A pointer is a variable used to store a *memory address*. Memory in a computer is made up of bytes arranged in a sequential manner. Each byte has a number associated with it which is called the address of the byte. The address values range from 0 to *one less* than the size of memory. For example, in 64MB of RAM, there are 64 x 2^{20} = 67,108,864 bytes. Therefore the addresses of these bytes will range from 0 to 67,108,863.

A variable of type `int` will occupy 4 bytes of memory. The compiler reserves 4 consecutive bytes from memory to store an integer value. The address of the first byte of these 4 allocated bytes is known as the address of the variable.

The Address Operator

To find the address of a variable, C provides an operator called the address operator (`&`). To find out the address of a variable, we need to place `&` in front of it. This program demonstrates how to use the address operator:

```c
#include<stdio.h>

int main()
{
    int i = 12;

    printf("Address of i = %u\n", &i);
    printf("Value of i = %d\n", i);

    return 0;
}
```

Expected Output

```
Address of i = 6356748
Value of i = 12
```

Declaring Pointer Variables

Just like any other variable you need to first declare a pointer variable before you can use it. Here is the syntax for declaring a pointer variable:

```
data_type *pointer_name;
```

Where `data_type` is the type of the pointer and `pointer_name` is the name of the variable, which can be any valid C identifier. Here are some examples for declaring pointers:

```
int *ip;
float *fp;
```

`int *ip` means that `ip` is a pointer variable that can *only store an address of a variable of type* `int`. Similarly, the pointer variable `fp` can only store the address of a variable of type `float`.

From now on, for convenience, we will refer to a *pointer variable* as a *pointer*.

Assigning an Address to a Pointer

After declaring a pointer the next step is to assign some valid memory address to it. You should *never* use a pointer variable without assigning a valid memory address to it, because, just after declaration it contains a random value and it may be pointing to *anywhere* in the memory. The use of an unassigned pointer may give an unpredictable result. It may even cause the program to crash. This example shows how to assign an address to a pointer:

```
int *ip, i = 10;
float *fp, f = 12.2;

ip = &i;
fp = &f;
```

Here `ip` is declared as a pointer to `int`, so it can only point to the memory address of an `int` variable. Similarly, `fp` can only point to the address of a `float` variable. In the last two statements, we have assigned the address of `i` and `f` to `ip` and `fp` respectively. It is important to note that if you assign the address of a `float` variable to a pointer to `int`, the compiler will not show you an error but the code may not produce the desired result.

Dereferencing a Pointer

Dereferencing a pointer simply means accessing data at the address stored in the pointer. Up until now, we have been using the name of the variable to access data inside it, but we can also access variable data *indirectly* using pointers. To do this a new operator called the *dereferencing operator* (`*`) is used. By placing the dereferencing operator before a pointer we can access the *value* of the variable whose address is stored in the pointer. Here is an example that shows the use of the dereferencing operator:

```c
#include<stdio.h>

int main()
{
    int i = 12;
    int *ip;

    ip = &i;

    printf("Address of i = %u\n", ip);
    printf("Value of i = %d\n", *ip);

    return 0;
}
```

Expected Output

```
Address of i = 6356748
Value of i = 12
```

The dereferencing operator can be read as “the value at the address”. For example, `*ip` can be read as “the value at address `ip`”.

Function Call by Reference

In this function calling method *addresses* of the actual arguments are copied and then assigned to the corresponding arguments in the function header. Now the calling and called functions are both using *pointers* that point to the *same data*. As a result, any changes made by the *called* function also affect the data values in the *calling function*. This program demonstrates call by reference:

```c
#include<stdio.h>

void change_values(int *xp, int *yp);

int main()
{
    int x = 10, y = 20;

    printf("Initial value of x in main() = %d\n", x);
    printf("Initial value of y in main() = %d\n\n", y);
```

```
    change_values(&x, &y);

    printf("Final value of x in main() = %d\n", x);
    printf("Final value of y in main() = %d\n\n", y);

    return 0;
}

void change_values(int *xp, int *yp)
{
    *xp += 1;
    *yp += 1;

    printf("Value of x in function = %d\n", *xp);
    printf("Value of y in function = %d\n\n", *yp);
}
```

Expected Output

```
Initial value of x in main() = 10
Initial value of y in main() = 20

Value of x in function = 11
Value of y in function = 21

Final value of x in main() = 11
Final value of y in main() = 21
```

In this code we are passing the *addresses* of integer variables to a function, so the arguments in the function header must be declared as a *pointer to* `int` which is written as `(int *)`. The expression `*xp += 1` means add **1** to the value at address `xp`.

When the function `change_values()` ends, control passes back to `main()` and the `printf()` statements print the new values of the integer variables `x` and `y`.

Exercise 10

Complete the following code so that it produces the expected output shown below.

```c
#include<stdio.h>

// Declare a function called my_sum
// to find the sum of two integers.
// It should accept 2 pointer to int arguments
// and return an integer answer.

int main()
{
    int x, a, b;

    printf("Enter the value for a: ");
    scanf("%d", &a);
    printf("Enter the value for b: ");
    scanf("%d", &b);

    // Call the function my_sum
    // to add a and b and assign the
    // answer to x.

    printf("\n\nThe sum of a and b = %d \n\n",x);

    return 0;
}

// Define the function my_sum.
```

Expected Output

```
Enter the value for a: 10
Enter the value for b: 12


The sum of a and b = 22
```

Arrays and Structures

What is an Array?

An array is a collection of *one or more* values of the *same type*. Each value is called an *element* of the array. The elements of the array share the same *variable name* but each element has its own unique *index* number. An array can be of any type, but, if an array is declared as a particular type, then *all* its elements must be of *the same type*. This is the syntax for declaring a one-dimensional array:

```
datatype array_name[size];
```

where `datatype` denotes the type of the elements in the array, `array_name` is the name of the array and must be a valid identifier, and `size` is the number of elements the array can hold. Variables and symbolic constants can be used to specify the size of an array when it is declared.

For example, to store the height of 100 students, we have to declare an array of size 100. This is the syntax for declaring an array called height which is of type float and has 100 elements:

```
float height[100];
```

In C, array indices start at 0, so `height[0]` is the first element, `height[1]` is the second element and so on. Note that the *last* element of the array will be `height[99]`.

Accessing Elements of an Array

An array index can be any expression that yields an integer value. For example:

```
int my_arr[5];
int i = 0, j = 2;

my_arr[i]; // 1st element
my_arr[i+1]; // 2nd element
my_arr[i+j]; // 3rd element
```

The first valid index (**0**) is known as the *lower bound*, while the last valid index is known as the *upper bound*. In the array `my_arr`, the last element is `my_arr[4]`. In these example lines of code, indexes **5**, **10** and **-1** are *not valid* but the C compiler will *not* give you an error message. Instead some *random value* will be printed. The C language doesn't check bounds of the array. It is the responsibility of the *programmer* to check array bounds whenever required.

```
printf("%d", my_arr[5]); // wrong
printf("%d", my_arr[10]); // wrong
printf("%d", my_arr[-1]); // wrong
```

Processing 1-D Arrays

This program uses **for** loops to take input and print elements of a 1-D array:

```c
#include<stdio.h>

int main()
{
    int arr[5], i;

    for(i = 0; i < 5; i++)
    {
        printf("Enter a[%d]: ", i);
        scanf("%d", &arr[i]);
    }

    printf("\nPrinting elements of the array: \n\n");

    for(i = 0; i < 5; i++)
    {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Expected Output

```
Enter a[0]: 10
Enter a[1]: 20
Enter a[2]: 30
Enter a[3]: 40
Enter a[4]: 50

Printing elements of the array:

10 20 30 40 50
```

In this code, we have declared an array of 5 integers and variable `i` of type **`int`**. Then a **`for`** loop is used to enter five elements into an array. In the **`scanf`**`()` statement we have used the address operator `(&)` on the *i*th element of the array: `arr[i]`. The second **`for`** loop prints the values of all the elements of the array, one by one, separated by a space.

Initializing an Array

When an array is declared inside a function the elements of the array have *random* values. If an array is global or static, then its elements are automatically initialized to **0**. This is the syntax for explicitly initializing elements of an array at the time of declaration:

```
datatype array_name[size] = {val1, val2, val3, ....., valN};
```

where `val1, val2 ..... valN` are constants known as *initializers*. Each value is separated by a *comma* and there must be a *semi-colon* after the closing curly brace. Here are some examples:

```
float temp[5] = {12.3, 4.1, 3.8, 9.5, 4.5}; // an array of 5 floats
int arr[] = {11, 22, 33, 44, 55, 66, 77, 88, 99}; // an array of 9 ints
```

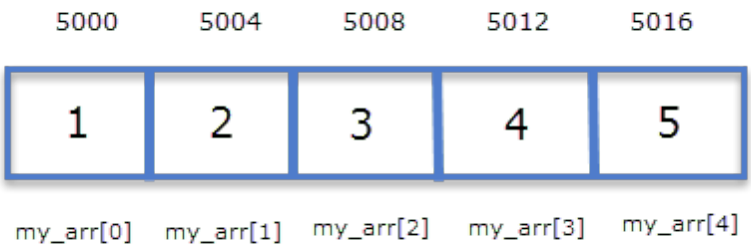
When initializing a 1-D array it is *optional* to specify the size of the array. If the number of initializers is *less than* the specified size, the *remaining* elements of the array are assigned a value of **0**. If the number of initializers is *greater than* the size of the array, the compiler will report an *error*.

Pointers and 1-D Arrays

In C, the elements of an array are stored in contiguous memory locations. For example: if we have this array declaration:

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

Then, the elements of `my_arr` are stored in memory like this:



Here the first element is at address **5000**. Since each integer occupies *4 bytes* the next element is at address **5004** and so on.

In C, pointers and arrays are very closely related. Behind the scenes, the compiler accesses elements of the array using pointer notation rather than index notation because accessing elements using a pointer

is very efficient when compared to index notation. The most important thing to remember about arrays in C is that:

> The name of an array is a *pointer* to the address of the *first element* of the array.

Using Pointers to Access Elements of an Array

You can easily access the values and addresses of the elements in an array using pointer arithmetic. Suppose `my_arr` is an array of 5 integers initialized using this statement:

```
int my_arr[5] = {11, 22, 33, 44, 55};
```

We have just learned that `my_arr` is a pointer to **`int`** or `(int *)`. So, in this example, `my_arr` points to the address of the first element of the array, `my_arr+1` points to the address of the second element, and so on. This means that:

`my_arr` is the same as `&my_arr[0]`
`my_arr + 1` is the same as `&my_arr[1]`
and so on

It also means that:

`*my_arr` is the same as `my_arr[0]`
`*(my_arr + 1)` is the same as `my_arr[1]`
and so on

Passing a 1-D Array to a Function

This example shows how to pass a one-dimensional array to a function in C:

```
#include<stdio.h>

void change_oned(int *my_array);

int main()
{
    int one_dim[] = {1,4,9,16,23}, i;

    printf("Original array: \n\n");
    for(i = 0; i < 5; i++)
    {
        printf("%d ", one_dim[i]);
    }
```

```
    change_oned(one_dim);

    printf("\n\nModified array: \n\n");
    for(i = 0; i < 5; i++)
    {
        printf("%d ", one_dim[i]);
    }

    return 0;
}

void change_oned(int *my_array)
{
    int i;

    for(i = 0; i < 5; i++)
    {
        my_array[i] += 5;
    }
}
```

Expected Output

```
Original array:

1 4 9 16 23

Modified array:

6 9 14 21 28
```

Since `one_dim` is a pointer to the first element of the array, we can pass `one_dim` to the function `change_oned()` *without using* the address operator, `&`. We are actually assigning the address of the first element in `one_dim` to the pointer `my_array`, which is of type `(int *)`. This means that we are using call by reference instead of call by value. So now both pointers, `one_dim` and `my_array`, point to the *same array*. Inside the function, we are using a **`for`** loop to increment every element of the array by 5. Since `one_dim` and `my_array` point to the *same data*, all the changes made here will affect the original array. Remember that you *don't need* to use the dereferencing operator when using array names.

The argument in the function header could also have been written as:

```
void change_oned(int my_array[5])
```

or

```
void change_oned(int my_array[])
```

Exercise 11

Complete the following code so that it produces the expected output shown below.

```
#include<stdio.h>

// Declare a function called my_array_sum
// to find the sum of an array of 5 integers.
// It should accept an array of 5 integers
// as an argument and return an integer answer.

int main()
{
    int i, x;

    // Declare an array called one_d that can hold 5 integers.

    for(i = 0; i < 5; i++)
    {
        printf("Enter the value for element %d of my_arr: ",i);
        scanf("%d", &one_d[i]);
    }

    // Call the function my_array_sum
    // to find the sum of the elements in my_arr
    // and assign the answer to x.

    printf("\n\nThe sum of the elements of the array = %d \n\n",x);

    return 0;
}

// Define the function my_array_sum.
```

Expected Output

```
Enter one_d[0]: 1
Enter one_d[1]: 2
Enter one_d[2]: 3
Enter one_d[3]: 4
Enter one_d[4]: 5


The sum of the elements of the array = 15
```

2-D Arrays

This is the syntax for declaring a 2-D array:

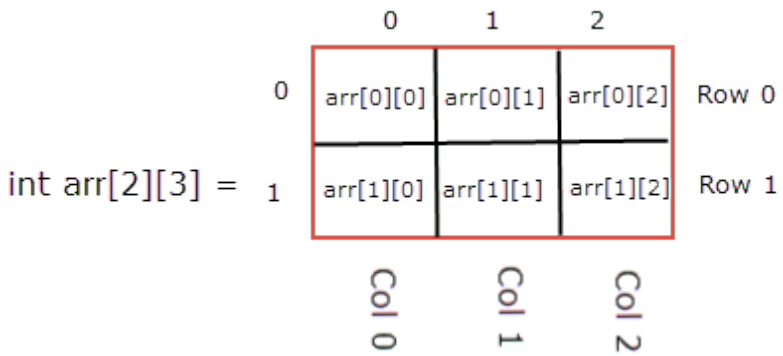
```
datatype array_name[ROW][COL];
```

Note that we just use two indices instead of one. The total number of elements in a 2-D array is `ROW*COL`.

Consider this example:

```
int arr[2][3];
```

This array can store 2x3 = 6 elements. You can visualize this 2-D array as a matrix of 2 rows and 3 columns using this diagram:



The individual elements of a 2-D array can be accessed by using *two indices* instead of one. The first index denotes the *row number* and the second denotes the *column number*. Both row and column indices start at 0.

Processing Elements of a 2-D Array

To process elements of a 2-D array, we can use *two nested* loops. In this example, the *outer* **`for`** loop loops through all the *rows* and the *inner* **`for`** loop loops through all the *columns*:

```
#include<stdio.h>

#define ROW 2
#define COL 2
```

```c
int main()
{
    int arr[ROW][COL], i, j;

    for(i = 0; i < ROW; i++)
    {
        for(j = 0; j < COL; j++)
        {
            printf("Enter arr[%d][%d]: ", i, j);
            scanf("%d", &arr[i][j]);
        }
    }

    printf("\nEntered 2-D array is: \n\n");

    for(i = 0; i < ROW; i++)
    {
        for(j = 0; j < COL; j++)
        {
            printf("%3d ", arr[i][j] );
        }
        printf("\n");
    }
    return 0;
}
```

Expected Output

```
Enter arr[0][0]: 1
Enter arr[0][1]: 2
Enter arr[1][0]: 3
Enter arr[1][1]: 4

Entered 2-D array is:

  1   2
  3   4
```

Initializing a 2-D Array

This example shows how to initialize a 2-D array:

```
int temp[2][3] = {
    {1, 2, 3},   // row 0
    {11, 22, 33} // row 1
};
```

This output shows the initial value of each element :

```
temp[0][0] = 1
temp[0][1] = 2
temp[0][2] = 3
temp[1][0] = 11
temp[1][1] = 22
temp[1][2] = 33
```

Pointer to an Array

We have seen that the name of an array is actually a point to the first element in the array. In C, we can also create a pointer that can point to the whole array instead of only one element of the array. This is known as a *pointer to an array*. This is the syntax for declaring a point to an array:

```
datatype (*array_name)[size];
```

For example, here is how you can declare a pointer named `array_p` that points to an array of 10 integers.

```
int (*array_p)[10]; \\ pointer to an array
```

In this case, the type of `array_p` is “pointer to an array of 10 integers”. Note that the parentheses are *necessary*. For example, this declaration:

```
int *array_p[10]; \\ wrong: array of pointers
```

means that `array_p` is an array of 10 elements where *each element* is of type “pointer to `int`” or `(int *)`.

Passing a 2-D Array to a Function

This example shows how to pass a 2-D array to a function in C:

```c
#include<stdio.h>

void change_twod(int (*my_array2)[3]);

int main()
{
    int i,j;
    int two_dims[2][3] = {
        {10,20,30},
        {45,55,65}
    };

    printf("Original array: \n\n");

    for(i = 0; i < 2; i++)
    {
        for(j = 0; j < 3; j++)
        {
            printf("%3d ", two_dims[i][j]);
        }
        printf("\n");
    }

    change_twod(two_dims);


    printf("\nModified array: \n\n");

    for(i = 0; i < 2; i++)
    {
        for(j = 0; j < 3; j++)
        {
            printf("%3d ", two_dims[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

```
void change_twod(int (*my_array2)[3])
{
    int i, j;

    for(i = 0; i < 2; i++)
    {
        for(j = 0; j < 3; j++)
        {
            my_array2[i][j] += 5;
        }
    }

}
```

Expected Output

```
Original array:

 10  20  30
 45  55  65

Modified array:

 15  25  35
 50  60  70
```

In C, 2-D arrays are stored in *row-major order* i.e. the first row is stored, then the second row is stored next to it, and so on. This means that a 2-D array is actually a 1-D array in which *each element* is also a *1-D array*. We know that the name of an array points to the first element of the array. We also know that, in a 2-D array, the *first element* is an *array*. Therefore, the name of a 2-D array is actually a *pointer to an array*.

In this example, the name of the 2-D array, `two_dims`, is a pointer to an *array of 3 integers*. The function `change_twod()` is called with the name of the 2D array, `two_dims`, as its argument. So `two_dims` is assigned to the pointer `my_array2` in the function call. Now both `two_dims` and `my_array2` point to the same 2-D array and, as a result, all the changes made inside the function will affect the original array.

The argument in the function header could also have been written as:

```
void change_twod(int my_array2[2][3])
```

or

```
void change_twod(int my_array2[][3])
```

Exercise 12

Complete the following code so that it produces the expected output shown below.

```
#include<stdio.h>

// Declare a function called my_2d_array_sum
// to find the sum of a 2x2 array of integers.
// It should accept a 2x2 array of integers
// as an argument and return an integer.

int main()
{
    int i, j, x;

    // Declare a 2x2 array of integers called two_d.

    for(i = 0; i < 2; i++)
    {
        for(j = 0; j < 2; j++)
        {
            printf("Enter two_d[%d][%d]: ",i,j);
            scanf("%d", &two_d[i][j]);
        }
    }

    // Call the function my_2d_array_sum
    // to find the sum of the elements in two_d
    // and assign the answer to x.

    printf("\n\nThe sum of the elements of the array = %d \n\n",x);

    return 0;
}

// Define the function my_2d_array_sum.
```

Expected Output

```
Enter two_d[0][0]: 1
Enter two_d[0][1]: 2
Enter two_d[1][0]: 3
Enter two_d[1][1]: 4


The sum of the elements of the array = 10
```

Structures

In C, *structures* can be used to create new data types. A structure allows related data, *of different types*, to be group together under a single name. Each data element of a structure is referred to as a *member* of the structure. The syntax for declaring a structure is:

```
struct tagname
{
    datatype member1;
    datatype member2;
    …
    datatype memberN;
};
```

where `tagname` is the name of the entire *type* of structure and `member1, member2, … memberN` are the members within the structure.

This is the syntax to create an *instance* of a structure:

```
struct tagname struct_name;
```

This is the syntax to access a member of the structure:

```
struct_name.member_name
```

Structure Example

This program shows how to declare a structure *type*, declare an *instance* of the structure and access *members* of the structure.

```c
#include <stdio.h>

struct mark_record
{
    int id;
    char *first_name;
    char *last_name;
    float mark;
};    // declare a structure type called mark_record

int main()
{
    struct mark_record student1;  // declare the variable student1
                                  // of type mark_record

    // assign values to the members of student1
    student1.id = 7803017;
    student1.first_name = "Mark";
    student1.last_name = "Pickering";
    student1.mark = 85.7;

    printf("%d", student1.id);
    printf("\t%s\t%s", student1.first_name, student1.last_name);
    printf("\t%5.2f\n", student1.mark);

    return 0;
}
```

Expected Output

```
7803017 Mark    Pickering       85.70
```

Array of Structures

Declaring an array of structure is the same as declaring an array of any other data type. Since an array is a collection of elements of the same type, in an array of structures, each element of an array is of the structure type. This example shows how to declare an array of structures:

```
struct mark_record student[50];
```

Subscript notation (`[]`) is used to access individual elements of the array and the dot operator (`.`) is used to access the members of each element as usual. This example shows how to print the `first_name` and `last_name` members of all the elements of the array using a **`for`** loop:

```
for(i = 0; i < 50; i++)
{
    printf("\t%s\t%s \n", student[i].first_name, student[i].last_name);
}
```

Pointer to a Structure

Just as we can have a pointer that points to the address of another variable of any data type, we can also have a pointer that can point to the *address of a structure*. This example shows how we can declare and initialize a pointer to a structure:

```
struct mark_record student_data;
struct mark_record *student = &student_data;
```

Note that a structure instance must *first be declared* when using this method so that the pointer can be initialised to point to its address.

This statement shows how to use the dereferencing operator (`*`) and dot operator (`.`) to access the structure members when using a pointer to the structure:

```
(*student).id = 7803017;
```

This method of accessing members of the structure, when using a pointer, is slightly confusing and not easy to read. So, C provides another way to access members using the *arrow operator* (`->`).

This statement shows the alternate, and *preferred*, arrow operator method for accessing a structure member when using a pointer:

```
student->id = 7803017;
```

Pointer to an Array of Structures

This example shows how we can declare and initialize a pointer to an array of structures:

```
struct mark_record student_data[50];
struct mark_record (*student)[50] = &student_data;
```

Again, note that an instance of the array of structures must *first be declared* when using this method so that the pointer can be initialised to point to its address.

This statement shows how to use the dereferencing operator (`*`) and dot operator (`.`) to access a member of the `i`th element of an array of structures when using a pointer to the array:

```
(*student)[i].id = 7803017;
```

Exercise 13

Complete the following code so that it produces the expected output shown below.

```c
#include <stdio.h>

// Declare a structure type called
// height_record with two members:
// an array called first_name with 50 elements of type char
// a float called height_in_cm

int main()
{

    // Declare an array of structures of type
    // height_record called height_data.
    // The array should have 2 elements.

    // Declare a pointer to an array of structures
    // of type height_record with 2 elements.
    // The pointer should be called student and be
    // initialized with the address of height_data.

    // Allow the user to enter the first name
    // and height for each element of the array.

    // Print out the first name and height
    // for each element of the array.

    return 0;
}
```

Expected Output

```
Student 1:
Enter first name: Mark
Enter height in cm: 178

Student 2:
Enter first name: Aidan
Enter height in cm: 185

Mark    178.0
Aidan   185.0
```