<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>String literals</td>
<td>19</td>
</tr>
<tr>
<td>Defining Constants</td>
<td>19</td>
</tr>
<tr>
<td>The #define Preprocessor</td>
<td>19</td>
</tr>
<tr>
<td>The const Keyword</td>
<td>20</td>
</tr>
<tr>
<td>C Storage Classes</td>
<td>22</td>
</tr>
<tr>
<td>The auto Storage Class</td>
<td>22</td>
</tr>
<tr>
<td>The register Storage Class</td>
<td>22</td>
</tr>
<tr>
<td>The static Storage Class</td>
<td>23</td>
</tr>
<tr>
<td>The extern Storage Class</td>
<td>23</td>
</tr>
<tr>
<td>C Operators</td>
<td>25</td>
</tr>
<tr>
<td>Arithmetic Operators</td>
<td>25</td>
</tr>
<tr>
<td>Relational Operators</td>
<td>26</td>
</tr>
<tr>
<td>Logical Operators</td>
<td>28</td>
</tr>
<tr>
<td>Bitwise Operators</td>
<td>29</td>
</tr>
<tr>
<td>Assignment Operators</td>
<td>31</td>
</tr>
<tr>
<td>Misc Operators ↦ sizeof &amp; ternary</td>
<td>33</td>
</tr>
<tr>
<td>Operators Precedence in C</td>
<td>33</td>
</tr>
<tr>
<td>Decision Making in C</td>
<td>35</td>
</tr>
<tr>
<td>if statement</td>
<td>36</td>
</tr>
<tr>
<td>Syntax</td>
<td>36</td>
</tr>
<tr>
<td>Flow Diagram</td>
<td>36</td>
</tr>
<tr>
<td>Example</td>
<td>36</td>
</tr>
<tr>
<td>if...else statement</td>
<td>37</td>
</tr>
<tr>
<td>Syntax</td>
<td>37</td>
</tr>
<tr>
<td>Flow Diagram</td>
<td>38</td>
</tr>
<tr>
<td>Example</td>
<td>38</td>
</tr>
<tr>
<td>The if...else if...else Statement</td>
<td>39</td>
</tr>
<tr>
<td>Syntax</td>
<td>39</td>
</tr>
<tr>
<td>Example</td>
<td>39</td>
</tr>
<tr>
<td>Nested if statements</td>
<td>40</td>
</tr>
<tr>
<td>Syntax</td>
<td>40</td>
</tr>
<tr>
<td>Example</td>
<td>40</td>
</tr>
<tr>
<td>switch statement</td>
<td>41</td>
</tr>
<tr>
<td>Syntax</td>
<td>41</td>
</tr>
<tr>
<td>Flow Diagram</td>
<td>42</td>
</tr>
<tr>
<td>Example</td>
<td>42</td>
</tr>
<tr>
<td>Nested switch statements</td>
<td>43</td>
</tr>
<tr>
<td>Syntax</td>
<td>43</td>
</tr>
<tr>
<td>Example</td>
<td>43</td>
</tr>
</tbody>
</table>
Installation on UNIX/Linux

If you are using Linux or UNIX, then check whether GCC is installed on your system by entering the following command from the command line:

```bash
$ gcc -v
```

If you have GNU compiler installed on your machine, then it should print a message something as follows:

```
Using built-in specs.
Target: i386-redhat-linux
Configured with: ../configure --prefix=/usr ........
Thread model: posix
gcc version 4.1.2 20080704 (Red Hat 4.1.2-46)
```

If GCC is not installed, then you will have to install it yourself using the detailed instructions available at http://gcc.gnu.org/install/

This tutorial has been written based on Linux and all the given examples have been compiled on CentOS flavor of Linux system.

Installation on Mac OS

To obtain GCC for Mac OS X, the easiest way to obtain GCC is to download the Xcode development environment from the Apple’s web site and follow the simple installation instructions. Once you have Xcode setup, you will be able to use GNU compiler for C/C++.

Xcode is currently available at developer.apple.com/technologies/tools/.

Installation on Windows

To install GCC at Windows you need to install MinGW. To install MinGW, go to the MinGW homepage, www.mingw.org, and follow the link to the MinGW download page. Download the latest version of the MinGW installation program, which should be named MinGW-<version>.exe.

While installing MinGW, at a minimum, you must install gcc-core, gcc-g++, binutils, and the MinGW runtime, but you may wish to install more.

Add the bin subdirectory of your MinGW installation to your PATH environment variable, so that you can specify these tools on the command line by their simple names.

When the installation is complete, you will be able to run gcc, g++, ar, ranlib, dlltool, and several other GNU tools from the Windows command line.
C Basic Syntax

This chapter will give details about all the basic syntax about C programming language including tokens, keywords, identifiers, etc.

You have seen a basic structure of C program, so it will be easy to understand other basic building blocks of the C programming language.

Tokens in C

A C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal or a symbol. For example, the following C statement consists of five tokens:

```c
printf("Hello, World! \n");
```

The individual tokens are:

```c
printf
{
 "Hello, World! \n"
}
;
```

Semicolons ;

In C program, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.

For example, following are two different statements:

```c
printf("Hello, World! \n");
return 0;
```
Comments

Comments are like helping text in your C program and they are ignored by the compiler. They start with /* and terminates with the characters */ as shown below:

/* my first program in C */

You cannot have comments within comments and they do not occur within a string or character literals.

Identifiers

A C identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore _ followed by zero or more letters, underscores, and digits (0 to 9).

C does not allow punctuation characters such as @, $, and % within identifiers. C is a case sensitive programming language. Thus, Manpower and manpower are two different identifiers in C. Here are some examples of acceptable identifiers:

mohd       zara       abc       move_name   a_123
myname50    _temp      j          a23b9       retVal

Keywords

The following list shows the reserved words in C. These reserved words may not be used as constant or variable or any other identifier names.

<table>
<thead>
<tr>
<th>auto</th>
<th>else</th>
<th>Long</th>
<th>switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>break</td>
<td>enum</td>
<td>register</td>
<td>typedef</td>
</tr>
<tr>
<td>case</td>
<td>extern</td>
<td>return</td>
<td>union</td>
</tr>
<tr>
<td>char</td>
<td>float</td>
<td>short</td>
<td>unsigned</td>
</tr>
<tr>
<td>const</td>
<td>for</td>
<td>signed</td>
<td>void</td>
</tr>
<tr>
<td>continue</td>
<td>goto</td>
<td>sizeof</td>
<td>volatile</td>
</tr>
<tr>
<td>default</td>
<td>if</td>
<td>static</td>
<td>while</td>
</tr>
<tr>
<td>do</td>
<td>int</td>
<td>struct</td>
<td>_packed</td>
</tr>
<tr>
<td>double</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
C Data Types

In the C programming language, data types refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

The types in C can be classified as follows:

<table>
<thead>
<tr>
<th>S.N.</th>
<th>Types and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Basic Types:</strong> They are arithmetic types and consists of the two types: (a) integer types and (b) floating-point types.</td>
</tr>
<tr>
<td>2</td>
<td><strong>Enumerated types:</strong> They are also arithmetic types and they are used to define variables that can only be assigned certain discrete integer values throughout the program.</td>
</tr>
<tr>
<td>3</td>
<td><strong>The type void:</strong> The type specifier <code>void</code> indicates that no value is available.</td>
</tr>
<tr>
<td>4</td>
<td><strong>Derived types:</strong> They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types.</td>
</tr>
</tbody>
</table>

The array types and structure types are referred to collectively as the aggregate types. The type of a function specifies the type of the function’s return value. We will see basic types in the following section, whereas, other types will be covered in the upcoming chapters.

Integer Types

Following table gives you details about standard integer types with its storage sizes and value ranges:

<table>
<thead>
<tr>
<th>Type</th>
<th>Storage size</th>
<th>Value range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Char</td>
<td>1 byte</td>
<td>-128 to 127 or 0 to 255</td>
</tr>
<tr>
<td>unsigned char</td>
<td>1 byte</td>
<td>0 to 255</td>
</tr>
</tbody>
</table>
10 = 20;
Floating-point literals

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

While representing using decimal form, you must include the decimal point, the exponent, or both and while representing using exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.14159</td>
<td>/* Legal */</td>
</tr>
<tr>
<td>314159E-5L</td>
<td>/* Legal */</td>
</tr>
<tr>
<td>510E</td>
<td>/* Illegal: incomplete exponent */</td>
</tr>
<tr>
<td>210f</td>
<td>/* Illegal: no decimal or exponent */</td>
</tr>
<tr>
<td>.e55</td>
<td>/* Illegal: missing integer or fraction */</td>
</tr>
</tbody>
</table>

Character constants

Character literals are enclosed in single quotes, e.g., 'x' and can be stored in a simple variable of char type.

A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

There are certain characters in C when they are preceded by a backslash they will have special meaning. \n, \t are used to represent like newline (\n) or tab (\t). Here, you have a list of some of such escape sequence codes:

<table>
<thead>
<tr>
<th>Escape sequence</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\</td>
<td>\ character</td>
</tr>
<tr>
<td>'</td>
<td>' character</td>
</tr>
<tr>
<td>&quot;</td>
<td>&quot; character</td>
</tr>
<tr>
<td>?</td>
<td>? character</td>
</tr>
<tr>
<td>a</td>
<td>Alert or bell</td>
</tr>
<tr>
<td>b</td>
<td>Backspace</td>
</tr>
<tr>
<td>f</td>
<td>Form feed</td>
</tr>
<tr>
<td>n</td>
<td>Newline</td>
</tr>
<tr>
<td>r</td>
<td>Carriage return</td>
</tr>
<tr>
<td>t</td>
<td>Horizontal tab</td>
</tr>
<tr>
<td>v</td>
<td>Vertical tab</td>
</tr>
<tr>
<td>\ooo</td>
<td>Octal number of one to three digits</td>
</tr>
</tbody>
</table>
Following is the example to show few escape sequence characters:

```c
#include <stdio.h>

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

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

```
Hello   World
```

**String literals**

String literals or constants are enclosed in double quotes "". A string contains characters that are similar to character constants: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separating them using whitespaces.

Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear"
"hello, \\
dear"
"hello, " "d" "ear"
```

**Defining Constants**

There are two simple ways in C to define constants:

1. Using `#define` preprocessor.
2. Using `const` keyword.

**The #define Preprocessor**

Following is the form to use `define` preprocessor to define a constant:
Example

```c
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 100;

    /* check the boolean condition */
    if( a < 20 )
    {
        /* if condition is true then print the following */
        printf("a is less than 20\n");
    }
    else
    {
        /* if condition is false then print the following */
        printf("a is not less than 20\n");
    }
    printf("value of a is : %d\n", a);
    return 0;
}
```

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

```
a is not less than 20;
value of a is : 100
```
The if...else if...else Statement

An if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement.

When using if, else if, else statements there are few points to keep in mind:

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax

The syntax of an if...else if...else statement in C programming language is:

```c
if (boolean_expression 1)
{
    /* Executes when the boolean expression 1 is true */
} else if( boolean_expression 2 )
{
    /* Executes when the boolean expression 2 is true */
} else if( boolean_expression 3 )
{
    /* Executes when the boolean expression 3 is true */
} else
{
    /* executes when the none of the above condition is true */
}
```

Example

```c
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 100;

    /* check the boolean condition */
    if( a == 10 )
    {
        /* if condition is true then print the following */
        printf("Value of a is 10\n");
    } else if( a == 20 )
    {
        /* if else if condition is true */
        printf("Value of a is 20\n");
    }
} 
```
```c
printf("Excellent!\n" );
break;
case 'B':
case 'C':
    printf("Well done\n" );
break;
case 'D':
    printf("You passed\n" );
break;
case 'F':
    printf("Better try again\n" );
break;
default :
    printf("Invalid grade\n" );
}
printf("Your grade is %c\n", grade );
return 0;
```

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

```
Well done
Your grade is B
```

### Nested switch statements

It is possible to have a switch as part of the statement sequence of an outer switch. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise.

### Syntax

The syntax for a **nested switch** statement is as follows:

```
switch(ch1) {
    case 'A':
        printf("This A is part of outer switch" );
    switch(ch2) {
        case 'A':
            printf("This A is part of inner switch" );
        break;
        case 'B': /* case code */
    } 
    break;
    case 'B': /* case code */
}
```

### Example

```c
#include <stdio.h>

int main ()
{
```
When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Unlike for and while loops, which test the loop condition at the top of the loop, the do...while loop in C programming language checks its condition at the bottom of the loop.

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

**Syntax**

The syntax of a **do...while** loop in C programming language is:

```
do
{ 
    statement(s);
} while ( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.
When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
```
What Are Pointers?

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address. The general form of a pointer variable declaration is:

```
type *var-name;
```

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk * you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

```
int *ip;  /* pointer to an integer */
double *dp; /* pointer to a double */
float *fp; /* pointer to a float */
char *ch; /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different types is the data type of the variable or constant that the pointer points to.

How to use Pointers?

There are few important operations, which we will do with the help of pointers very frequently. (a) we define a pointer variable (b) assign the address of a variable to a pointer and (c) finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations:

```c
#include <stdio.h>

int main ()
{
    int var = 20;  /* actual variable declaration */
    int *ip;  /* pointer variable declaration */

    ip = &var;  /* store address of var in pointer variable*/

    printf("Address of var variable: %x\n", &var);

    /* address stored in pointer variable */
    printf("Address stored in ip variable: %x\n", ip);

    /* access the value using the pointer */
    printf("Value of *ip variable: %d\n", *ip);

    return 0;
}
```
ptr++

Now, after the above operation, the `ptr` will point to the location 1004 because each time `ptr` is incremented, it will point to the next integer location which is 4 bytes next to the current location. This operation will move the pointer to next memory location without impacting actual value at the memory location. If `ptr` points to a character whose address is 1000, then above operation will point to the location 1001 because next character will be available at 1001.

### Incrementing a Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array:

```c
#include <stdio.h>

const int MAX = 3;
int main ()
{
    int var[] = {10, 100, 200};
    int i, *ptr;
    /* let us have array address in pointer */
    ptr = var;
    for (i = 0; i < MAX; i++)
    {
        printf("Address of var[%d] = %x\n", i, ptr);
        printf("Value of var[%d] = %d\n", i, *ptr);
        /* move to the next location */
        ptr++;
    }
    return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
```
When the above code is compiled and executed, it produces the following result:

| Number of seconds : 1294450468 |

The function, which can accept a pointer, can also accept an array as shown in the following example:

```c
#include <stdio.h>
/* function declaration */
double getAverage(int *arr, int size);

int main ()
{
    /* an int array with 5 elements */
    int balance[5] = {1000, 2, 3, 17, 50};
    double avg;

    /* pass pointer to the array as an argument */
    avg = getAverage( balance, 5 );

    /* output the returned value */
    printf( "Average value is: %f
\n", avg );
    return 0;
}

double getAverage(int *arr, int size)
{
    int i, sum = 0;
    double avg;
    for (i = 0; i < size; ++i)
    {
        sum += arr[i];
    }
    avg = (double)sum / size;
    return avg;
}
```

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

| Average value is: 214.40000 |

Return pointer from functions

As we have seen in last chapter how C programming language allows to return an array from a function, similar way C allows you to return a pointer from a function. To do so, you would have to declare a function returning a pointer as in the following example:

```c
int * myFunction()
{
```
1421301276
930971084
123250484
106932140
1604461820
149169022
*(p + [0]) : 1523198053
*(p + [1]) : 1187214107
*(p + [2]) : 1108300978
*(p + [3]) : 430494959
*(p + [4]) : 1421301276
*(p + [5]) : 930971084
*(p + [6]) : 123250484
*(p + [7]) : 106932140
*(p + [8]) : 1604461820
*(p + [9]) : 149169022
C Structures

C arrays allow you to define type of variables that can hold several data items of the same kind but structure is another user defined data type available in C programming, which allows you to combine data items of different kinds.

Structures are used to represent a record, suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:

- Title
- Author
- Subject
- Book ID

Defining a Structure

To define a structure, you must use the `struct` statement. The `struct` statement defines a new data type, with more than one member for your program. The format of the `struct` statement is this:

```
struct [structure tag]
{
    member definition;
    member definition;
    ...
    member definition;
} [one or more structure variables];
```

The structure tag is optional and each member definition is a normal variable definition, such as `int i;` or `float f;` or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure:

```
struct Books
{
```

Preview from Notesale.co.uk
Page 103 of 146
When the above code is compiled and executed, it produces the following result:

<table>
<thead>
<tr>
<th>Book title</th>
<th>C Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Book author</td>
<td>Nuha Ali</td>
</tr>
<tr>
<td>Book subject</td>
<td>C Programming Tutorial</td>
</tr>
<tr>
<td>Book book_id</td>
<td>6495407</td>
</tr>
<tr>
<td>Book title</td>
<td>Telecom Billing</td>
</tr>
<tr>
<td>Book author</td>
<td>Zara Ali</td>
</tr>
<tr>
<td>Book subject</td>
<td>Telecom Billing Tutorial</td>
</tr>
<tr>
<td>Book book_id</td>
<td>6495700</td>
</tr>
</tbody>
</table>

Pointers to Structures

You can define pointers to structures in very similar way as you define pointer to any other variable as follows:

```c
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the & operator before the structure's name as follows:

```c
struct_pointer = &Book1;
```

To access the members of a structure using a **pointer** to that **structure**, you must use the `->` operator as follows:

```c
struct_pointer->title;
```

Let us re-write above example using structure pointer, hope this will be easy for you to understand the concept:

```c
#include <stdio.h>
```
Bit Fields

Suppose your C program contains a number of TRUE/FALSE variables grouped in a structure called status, as follows:

```c
struct {
    unsigned int widthValidated;
    unsigned int heightValidated;
} status;
```

This structure requires 8 bytes of memory space but in actual we are going to store either 0 or 1 in each of the variables. The C programming language offers a better way to utilize the memory space in such situation. If you are using such variables inside a structure then you can define the width of a variable which tells the C compiler that you are going to use only those number of bits. For example, above structure can be re-written as follows:

```c
struct {
    unsigned int widthValidated : 1;
    unsigned int heightValidated : 1;
} status;
```

Now, the above structure will require 4 bytes of memory space for status variable but only 2 bits will be used to store the values. If you will use up to 32 variables each one with a width of 1 bit, then also status structure will use 4 bytes, but as soon as you will have 33 variables, then it will allocate next slot of the memory and it will start using 64 bytes. Let us check the following example to understand the concept:

```c
#include <stdio.h>
#include <string.h>
/* define simple structure */
struct {
    unsigned int widthValidated;
    unsigned int heightValidated;
} status1;

/* define a structure with bit fields */
```
When the above code is compiled and executed, it produces the following result:

```
File : test.c
Date : Jun 2 2012
Time : 03:36:24
Line : 8
ANSI : 1
```

Preprocessor Operators

The C preprocessor offers following operators to help you in creating macros:

**Macro Continuation (\)**

A macro usually must be contained on a single line. The macro continuation operator is used to continue a macro that is too long for a single line. For example:

```
#define message_for(a, b) \
    printf(#a " and " #b " : We love you!\n")
```

**Stringize (#)**

The stringize or number-sign operator ("#"), when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro that has a specified argument or parameter list. For example:

```
#include <stdio.h>
#define message_for(a, b) \
    printf(#a " and " #b " : We love you!\n")

int main(void)
{
    message_for(Carole, Debra);
    return 0;
}
```

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

The Integer promotion is the process by which values of integer type "smaller" than int or unsigned int are converted either to int or unsigned int. Consider an example of adding a character in an int:

```c
#include <stdio.h>

main()
{
    int i = 17;
    char c = 'c'; /* ascii value is 99 */
    int sum;
    sum = i + c;
    printf("Value of sum : %d\n", sum);
}
```

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

```
Value of sum : 116
```

Here, value of sum is coming as 116 because compiler is doing integer promotion and converting the value of 'c' to ascii before performing actual addition operation.

Usual Arithmetic Conversion

The usual arithmetic conversions are implicitly performed to cast their values in a common type. Compiler first performs integer promotion, if operands still have different types then they are converted to the type that appears highest in the following hierarchy:
Command Line Arguments

It is possible to pass some values from the command line to your C programs when they are executed. These values are called command line arguments and many times they are important for your program specially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main() function arguments where argc refers to the number of arguments passed, and argv[] is a pointer array which points to each argument passed to the program. Following is a simple example which checks if there is any argument passed from the command line and take action accordingly:

```c
#include <stdio.h>

int main (int argc, char *argv[]) {
    if (argc == 2) {
        printf("The argument supplied is %s\n", argv[1]);
    } else if (argc > 2) {
        printf("Too many arguments supplied.\n");
    } else {
        printf("One argument expected.\n");
    }
}
```

When the above code is compiled and executed with a single argument, it produces the following result.

```
$./a.out testing
The argument supplied is testing
```

When the above code is compiled and executed with a two arguments, it produces the following result.