C/C++ programming

C is a middle level language. As a middle level language, C allows the manipulation of bits, bytes and addresses (well suited for system-level programming) – the basic elements with which the computer functions. Despite this fact C code is very portable. Portability means that it is easy to adapt software written for one type of computer or operating system to another. For example, you can easily convert a program written for DOS so that it runs under Windows, that program is portable.

|  |  |
| --- | --- |
| High Level | Ada, Modula-2, Pascal, COBOL, FORTRAN, BASIC |
| Middle Level | Java, C++, C, FORTH |
| Lowest Level | Macro-assembler, Assembler |

All high-level programming languages support the concept of data types.

A ***data type*** defines a set of values that a variable can store along with a set of operations that can be performed on that variable.

Common data types are integer, character, and real.

**Note: C is not a strongly typed language, as are Pascal and Ada. C permits almost all type conversions. For Example, you may free intermix character and integer types in an expression.**

**Unlike a high level language, C performs almost no runtime error checking. For Example, no check is performed to ensure that array boundaries are not overrun. These types of checks are the responsibility of the programmer.**

C is a structured language is not, technically, a block-structured language.

|  |  |
| --- | --- |
| Block structure language | Block structures languages permit procedures or functions to be declared inside other procedures or functions. Since C does not allow the creation of functions within functions, it cannot formally be called block-structured. |
| Structure language | The distinguishing feature of a structure language is compartmentalization of code and data. This is the ability of a language to section off and hide from the rest of the program all information and instructions necessary to perform a specific task.  A structured language allows you a variety of programming possibilities. It directly supports several loop constructs, such while, do-while, and for. In structure language, the use of goto is either prohibited or discouraged and is not the common form of program control. |

The 32 keywords of C language ( 27 were defined by the original version of C. Five were added by the ANSI C committee - enum, const, signed, void and volatile)

**Note: all C and C++ keywords are in lower case.**

|  |  |  |  |
| --- | --- | --- | --- |
| **32 keywords defined by standard 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 |

**Note:** neither C nor C++ provides any keyword that performs such things as input/output (I/O) operations, high level mathematical computations, or character handling. As a result, most programs include calls to various functions contained in the ***standard library***.

All C++ compilers come with a standard library of functions that perform most commonly needed tasks.

**Expression:** expressions are formed from atomic elements: data and operations. Data may be represented either by variables or by constants.

**There are five atomic data types in C**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Character (**char**) | Integer (**int**) | Floating-point (**float**) | Double floating-point (**double**) | Value less (**void**) |

All other data types in C are based upon one of these types.

**Note:** The size and range of these data types may vary between processor types and compilers. However, in all cases a character is 1 byte.

The size of an integer is usually the same as the word length of the execution environment of the program. For most 16-bit environments, such as DOS or Windows 3.1, an integer is 16 bits. For most 32-bit environments, such as Windows NT, an integer is 32 bits.

Both C and C++ only stipulate the ***minimum range*** of each data type, not its size in bytes.

**Note: To the five basic data types defined by C, C++ add two more: bool and wchar\_t**

**Standard C++ does not specify a minimum size or range for the basic types. Instead it simply states that they must meet the certain requirements. For example, standard C++ states that an int will “have the natural size suggested by the architecture of the execution environment.” In all cases this will meet or exceed the minimum ranges specified by Standard C.**

**Each C++ compiler specified the size and range of the basic types in the header <climits>**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Type** | **Typical size in bits** | **Minimal range** | **Comments** |  |
| char | 8 | -127 to 127 | Value of type char are generally used to hold values defined by the ASCII character set |  |
| unsigned char | 8 | 0 to 255 |  |  |
| signed char | 8 | -127 to 127 |  |  |
| int | 16 or 32 | -32767 to 32767 | Integers will generally correspond to the natural size of a word on the host computer |  |
| unsigned int | 16 or 32 | 0 to 65535 |  |  |
| signed int | 16 or 32 | Same as int |  |  |
| short int | 16 | -32767 to -32767 |  |  |
| unsigned short int | 16 | 0 to 65535 |  |  |
| signed short int | 16 | Same as short int |  |  |
| long int | 32 | -2147483647 to -2147483647 |  |  |
| signed long int | 32 | Same as long int |  |  |
| unsigned long int | 32 | 0 to 4294967295 |  |  |
| float | 32 | Six digits of precision | Exact format of floating-point values will depend upon how they are implemented. | The range of float and double will depend upon the method used to represent the floating point numbers |
| double | 64 | Ten digits of precision |  |  |
| long double | 80 | Ten digits of precision |  |  |
| void |  |  | Void either explicitly declares a function returning no value or creates generic pointers. |  |

Note: values outside that range may be handled differently by different compilers.

**Modifying the basic Types**

Except for type void, the basic data types may have various modifiers preceding them. Modifiers can be used to alter the meaning of the base type to fit various situations more precisely.

|  |
| --- |
| **List of modifiers** |
| signed |
| unsigned |
| long |
| short |
|  |

Note: The use of signed on integers is allowed, but redundant because the default integer declaration assumes a signed number. The most important use of signed is to modify char in implementations in which char is unsigned by default.

The difference between signed and unsigned integers is in the way that the high order bit if the integer is interpreted. If you specify a signed integer, the compiler generates the code that assumes that the high-order bit of an integer is to be used as a ***sign flag***.

If the sign flag is 0, the number is positive; if it is 1, the number is negative.

In general, negative numbers are represented using the two’s complement approach, which reverses all bits in the number (except the sign flag), adds 1 to this number, and set the sign flag 1.

Example:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Type | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 | 2048 | 4096 | 8192 | 16384 | Sign flag | No |
|  | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |  |
| Signed int | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  | -1 |
| Unsigned int | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 65535 |

***Identifiers:*** *in C/C++, the names of variables, functions, labels, and various other user-defined objects are called identifiers.*

*Note: The first characters must be a letter or underscore and subsequent characters must be either letters, digits, or underscores.*

***Correct incorrect***

*Count 1count*

*test23 hi!there*

*high\_balance high…balance*

In C, identifiers may be of any length. If the identifier will be involved in an external link process, then at least the first six characters will be significant. These identifiers, called ***external names***, including function names and global variables that are shared between files.

If the identifier is not used in an external link process, then at least the first 31 characters will be significant. This type of identifier is called an ***internal name*** and includes the name of local variables.

In C++, there is no limit to the length of an identifier, and at least the first 1024 characters are significant.

In an identifier, upper and lowercase are treated as distinct. Hence count, Count and COUNT are three separates identifiers.

An identifier cannot be the same as a C or C++ keyword, and should not have the same name as functions that are in the C or C++ library.

**Variables:** local variables, formal parameters and global variables

**Local variable:** variables that are declared inside a function are called local variables. These variables are referred to as ***automatic*** variables.

The C language contains the keyword **auto**, which can be used to declare local variables. However, since all non-global variables are, by default, assumed to be **auto**. This keyword virtually never used. It has been said that **auto** was included in C to provide for source-level compatibility with its predecessor B. Further, **auto** is supported in C++ to provide compatibility with C.

There is an important difference between C and C++ as to where you can declare local variables. In C, you must declare local variables at the start of the block in which they are defined, prior to any “action” statement.

Local variables are stored on the stack.

**Formal Parameters**: If a function is to use arguments, it must declare variables that will accept the values of the arguments. These variables are called the formal parameters of the function. As local variables, they are also dynamic and are destroyed upon exit from the function.

**Global Variables:** global variables are known throughout the program and may be used by any peace of code. If a global variable and a local variable have the same name, all references to that variable name inside the code block in which the local variable is declared will refer to that local variable and have no effect on the global variable.

Storage for global variables is in a fixed region of memory set aside for this purpose by the compiler.

**Access Modifiers**

**Access Modifiers**: There are two modifiers that control how variables may be accessed or modified. These qualifies are **const** and **volatile**.

**const:** The compiler is free to place variables of this type into read-only memory(ROM)

**Note**: A variable of type **const** can be modified by something outside your program. For example , a hardware device may set its value. However, by declaring a variable as **const,** you can prove that any changes to that variable occur because of external events.

**volatile:**  the modifier volatile tells the compiler that a variable’s value may be changed in way not explicitly specified by the program. For example, a global variable’s address may be passes to the operating system’s clock routine and used to hold the real time of the system. In this situation, the contents of variable are altered without any explicit assignment statements in the program.

This is important because most C/C++ compilers automatically optimize certain expressions by assuming that a variable’s content is unchanging if it does not occur on the left side of an assignment statement; thus, it might not be reexamined each time it is referenced. Also, some compilers change the order of evaluation of an expression during the compilation process. The **volatile** modifier prevents these changes.

Note: you can use const and volatile together. For example, if 0x30 is assumed to be the value of a port that is changed by external conditions only, the following declaration would prevent any possibility of accidental side effects:

const volatile char\* port = (const volatile char\*) 0x30;

**Storage class specifiers**

There are four storage class specifiers supported by C:

|  |  |  |
| --- | --- | --- |
| 1 | extern | The extern specifier tells the compiler that the variable types and names that follow it have been defined elsewhere. extern let the compiler know what the types and names are for these global variables without actually creating storage for them again. When the linker links the two modules, all references to the external variables are resolved. |
| 2 | static | static variables are permanent variables within their own function or file. |
| 3 | register | The register specifier requested that the compiler keep the value of a variable in a register of the CPU rather than in memory. This meant the operations on a register variable could occur much faster than on a normal variable because the register variable was actually held in the CPU and did not require a memory access to determine or modify its value.  **Note:** You can only apply the register specifier to local variables and to the formal parameters in a function. Global register variables are not allowed.  In C you cannot find address of register variable using & operator but this restriction does not apply on C++. |
| 4 | auto |  |

General form of a declaration that uses one is shown below:

***storage\_specifier type******var\_name;***

**wide character:**

wide characters are 16 bit long. In C, this type is defined in aheader file and is not a build-in type. In C++ wchar\_t is built in. To specify a wide character constant, precede the character with L

wchar\_t wc;

wc = L’A’;

|  |
| --- |
| **Tips** |
| If you follow the number with an F, the number is trated as a float. If you follow it with an L, the number becomes a long double. For integer types, the U suffix stands for unsigned and the L for long. |
| C/C++ allows you to specify integer constants in hexadecimal or octal instead of decimal. A hexadecimal constant must consist of a 0x followed by the constant in hexadecimal form. An octal constant begins with a 0.  int hex = 0x80; /\* 128 in decimal \*/  int oct = 012; /\* 10 in decimal\*/ |

**Backslash Character Constants/ escape sequences**

Backslash(\) character is a special character that can be used to enter special characters as constants.

|  |  |
| --- | --- |
| **Backslash codes** | |
| **Code** | **Meaning** |
| **\b** | **Backspace -** Moves the active position to the previous position on the current line. If the active position is at the initial position of a line, the behavior of the display device is unspecified.    **Example**  printf("This is a test\n");  printf("This is\b a test\n");  **output:**    printf("This is\b a test\n");  Cursor comes back to 1 character (on s) and print “ a test” |
| **\f** | **Form feed –** It skips to the start of the next page. (Applies mostly to terminals where the output device is a printer rather than a VDU.)  **Example**  printf("\fform feed \n");  **output:** |
| **\n** | **New line -** Moves the active position to the initial position of the next line.  **Example**  printf("\nNew file");  **output** |
| **\r** | **Carriage return** - to come in start of line. Moves the active position to the initial position of the current line.  The name comes from a printer's carriage, as monitors were rare when the name was coined. This is commonly escaped as "\r", abbreviated CR, and has ASCII value 13 or 0x0D.  Originally, the term "carriage return" referred to a mechanism or lever on a [typewriter](https://en.wikipedia.org/wiki/Typewriter). It was used after typing a line of text and caused the assembly holding the paper (the carriage) to return to the right so that the machine was ready to type again on the left-hand side of the paper. The lever would also usually advance the paper to the next line.  **Example**  printf("Carriage\r ret \n");  **output** |
| **\t** | **Horizontal tab**  **Example**  printf("\tHorizontal tab\n");  **output** |
| **\”** | **Double quote**  **Example**  printf("\"Double quote\n");  **output** |
| **\’** | **Single quote**  **Example**  printf("\'Single quote\n"); |
| **\0** | **Null - 0(zero) – it is null character**  **Example**  printf("Null char\0 remove this\n"); |
| **\\** | **Backslash**  printf("\\Backslash\n"); |
| **\v** | **Vertical Tab**  printf("\tVertical tab\n"); |
| **\a** | **Alert**  printf("\aalert\n");  This will produce a beep sound |
| **\?** | **Question mark**  printf("\?question mark\n"); |
| **\N** | **Octal constant (where N is an octal constant)**  **TODO- need to understand and write example** |
| **\xN** | **Hexadecimal constant( where N is a hexadecimal constant)**  **TODO- need to understand and write example** |

Numeral systems conversion table

|  |  |  |  |
| --- | --- | --- | --- |
| Decimal  Base-10 | Binary  Base-2 | Octal  Base-8 | Hexadecimal  Base-16 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| 2 | 10 | 2 | 2 |
| 3 | 11 | 3 | 3 |
| 4 | 100 | 4 | 4 |
| 5 | 101 | 5 | 5 |
| 6 | 110 | 6 | 6 |
| 7 | 111 | 7 | 7 |
| 8 | 1000 | 10 | 8 |
| 9 | 1001 | 11 | 9 |
| 10 | 1010 | 12 | A |
| 11 | 1011 | 13 | B |
| 12 | 1100 | 14 | C |
| 13 | 1101 | 15 | D |
| 14 | 1110 | 16 | E |
| 15 | 1111 | 17 | F |
| 16 | 10000 | 20 | 10 |
| 17 | 10001 | 21 | 11 |
| 18 | 10010 | 22 | 12 |
| 19 | 10011 | 23 | 13 |
| 20 | 10100 | 24 | 14 |
| 21 | 10101 | 25 | 15 |
| 22 | 10110 | 26 | 16 |
| 23 | 10111 | 27 | 17 |
| 24 | 11000 | 30 | 18 |
| 25 | 11001 | 31 | 19 |
| 26 | 11010 | 32 | 1A |
| 27 | 11011 | 33 | 1B |
| 28 | 11100 | 34 | 1C |
| 29 | 11101 | 35 | 1D |
| 30 | 11110 | 36 | 1E |
| 31 | 11111 | 37 | 1F |
| 32 | 100000 | 40 | 20 |

**Tips**

Constants refer to fixed values that the program may not alter during its execution. These fixed values are also called **literals**.