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:

For example, here is 32,767:

0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

If the high-order bit were set to 1, the number would be interpreted as −1.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Type | Sign flag  32768 | 16384 | 8192 | 4096 | 2048 | 1024 | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | No |
| signed int | 0 (positive number) | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 32767 |
| Set higher bit 1 | 1 (negative number) | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |  |
| Higher bit is 1 so number is negative. To know the negative number first reverse evert bit except sign flag and then add 1 | | | | | | | | | | | | | | | | | |
|  | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| Add 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | -1 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

***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**.

**Note** relational and logical operators always produce a result that is either

true or false

|  |  |
| --- | --- |
| **Relation operators** | |
| > | Greater than |
| >= | Greater than or equal |
| < | Less than |
| <= | Less than or equal |
| = = | Equal |
| != | Not equal |
| **Logical operators** | |
| && | AND |
| || | OR |
| ! | NOT |
| **Bitwise Operators** | |
| & | AND |
| | | OR |
| ^ | Exclusive OR (XOR) |
| ~ | One's complement (NOT) |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **P** | **Q** | **P & Q** | **P|Q** | **P^Q** | **~P** |
| **0** | **0** | **0** | **0** | **0** | **1** |
| **0** | **1** | **0** | **1** | **1** | **1** |
| **1** | **0** | **0** | **1** | **1** | **0** |
| **1** | **1** | **1** | **1** | **0** | **0** |

**Bit shift operator**

The bit-shift operators, >> and <<, move all bits in a value to the right or left as

specified. The general form of the shift-right statement is

*value >> number of bit positions*

The general form of the shift-left statement is

*value << number of bit positions*

As bits are shifted off one end, 0's are brought in the other end. (In the case of a

signed, negative integer, a right shift will cause a 1 to be brought in so that the sign bit

is preserved.) Remember, a shift is not a rotate. That is, the bits shifted off one end do

not come back around to the other. The bits shifted off are lost.

|  |  |  |
| --- | --- | --- |
| **Left shift** | | |
| **unsigned char x;** | **Executes** | **Value of x** |
| **X = 15;** | **00001111** | **15** |
| **X<<1** | **00011110** | **30** |
| **X<<1** | **00111100** | **60** |
| **X<<1** | **01111000** | **120** |
| **X<<1** | **11110000** | **240** |
| **X<<1** | **11100000** | **224** |
| **X<<1** | **11000000** | **192** |
| **X<<1** | **10000000** | **128** |
| **X<<1** | **00000000** | **0** |
| **X<<1** | **00000000** | **0** |
| **signed char x;** |  |  |
| **x = 15;** | **00001111** | **15** |
| **x<<1** | **00011110** | **30** |
| **x<<1** | **00111100** | **60** |
| **x<<1** | **01111000** | **120** |
| **x<<1** | **11110000** | **-16 – it is negative number**  **11110000**  **Get the negative number**  **Step1: make reverse every bit except signed bit**  **10001111**  **Step 2:Add 1 in this**  **10010000 = -16** |
| **x<<1** | **11100000** | **-32**  **Step 1: 10011111**  **Step 2: 10100000** |
| **x<<1** | **11000000** | **-128**  **Step 1: 10111111**  **Step 2: 11000000** |
| **x<<1** | **10000000** | **0**  **Step 1: 11111111**  **Step 2: 10000000** |
| **x<<1** | **00000000** | **0** |
| **x<<1** | **00000000** | **0** |
| **unsigned char x;** |  |  |
| **x = 15;** | **00001111** | **15** |
| **x>>1** | **00000111** | **7** |
| **x>>1** | **00000011** | **3** |
| **x>>1** | **00000001** | **1** |
| **x>>1** | **00000000** | **0** |
| **x>>1** | **00000000** | **0** |
| **signed char x;** |  |  |
| **x = 15;** | **00001111** | **15** |
| **x>>1** | **00000111** | **7** |
| **x>>1** | **00000011** | **3** |
| **x>>1** | **00000001** | **1** |
| **x>>1** | **00000000** | **0** |
| **x>>1** | **00000000** | **0** |

**ternary operator ?**

**Expression:** *Exp1 ? Exp2 : Exp3;*

The Compile-Time Operator sizeof

**sizeof** is a unary compile-time operator that returns the length, in bytes, of the variable

or parenthesized type- specifier that it precedes. For example, assuming that integers

are 4 bytes and **double**s are 8 bytes,

double f;

printf("%d ", sizeof(char));

printf("%d ", sizeof f);

printf("%d", sizeof(int));

will display 1 **8 4**.

Remember, to compute the size of a type, you must enclose the type name in parentheses.

This is not necessary for variable names, although there is no harm done if you do so.

The Comma Operator

The left side of the comma operator is always evaluated as **void**. This means that the expression on the right side becomes the value of the total comma-separated expression.

For example,

int y =1;

x = (y=3, y+1);

first assigns **y** the value 3 and then assigns **x** the value 4.V**alue of x will become 4**

The parentheses are necessary because the comma operator has a lower precedence than the assignment operator.

Precedence Summary

Table 2-8 lists the precedence of all operators defined by C. Note that all operators,

except the unary operators and **?**, associate from left to right. The unary operators

(**\***, **&**, ) and **?** associate from right to left.

Order of Evaluation

Neither C nor C++ specifies the order in which the subexpressions of an expression are

evaluated. This leaves the compiler free to rearrange an expression to produce more

optimal code. However, it also means that your code should never rely upon the order

in which subexpressions are evaluated. For example, the expression

x = f1() + f2();

does not ensure that **f1( )** will be called before **f2( )**.

Type Conversion in Expressions

When constants and variables of different types are mixed in an expression, they are

all converted to the same type. The compiler converts all operands up to the type of

the largest operand, which is called *type promotion*. First, all **char** and **short int** values

are automatically elevated to **int**. (This process is called *integral promotion*.)

Compound Assignments

There is a variation on the assignment statement, called *compound assignment*, that

simplifies the coding of a certain type of assignment operation. For example,

x = x+10;

can be written as

x += 10;

Compound assignment is also commonly referred to as *shorthand assignment* because it is more compact.

Nested ifs

**Note:** The C language guarantees at least 15 levels of nesting. In practice, most compilers allow substantially more. More importantly, Standard C++ suggests that at least 256 levels of nested **if**s be allowed in a C++ program. However, nesting beyond a few levels is seldom necessary, and excessive nesting can quickly confuse the meaning of an algorithm.

Switch

In C, a **switch** can have at least 257 **case** statements. Standard C++ recommends that *at least* 16,384 **case** statements be supported! In practice, you will want to limit the number of **case** statements to a smaller amount for efficiency.

The Infinite Loop

for( ; ; ) printf("This loop will run forever.\n");

Actually, the **for(;;)** construct does not guarantee an infinite loop because a **break**

statement, encountered anywhere inside the body of a loop, causes immediate

termination.

ch = '\0';

for( ; ; ) {

ch = getchar(); /\* get a character \*/

if(ch=='A') break; /\* exit the loop \*/

}

for Loops with No Bodies

A statement may be empty. This means that the body of the **for** loop (or any other loop) may also be empty. You can use this fact to improve the efficiency of certain algorithms and to create time delay loops.

for( ; \*str == ' '; str++) ;

Array

An *array* is a collection of variables of the same type that are referred to through a common name

A specific element in an array is accessed by an index. In C/C++, all arrays consist of contiguous memory locations.

Single-Dimension Arrays

*type var\_name[size]*;

double balance[100];

balance[3] = 12.23;

The amount of storage required to hold an array is directly related to its type and size. For a single-dimension array, the total size in bytes is computed as shown here:

total bytes = sizeof(base type) x size of array

C/C++ has no bounds checking on arrays. You could overwrite either end of an array and write into some other variable's data or even into the program's code. As the programmer, it is your job to provide bounds checking where needed.

Single-dimension arrays are essentially lists of information of the same type that are stored in contiguous memory locations in index order.

Generating a Pointer to an Array

You can generate a pointer to the first element of an array by simply specifying the array name, without any index. For example,

int \*p;

int sample[10];

p = sample;

**sample** and **&sample[0]** both produce the same results. However, in professionally written C/C++ code, you will almost never see **&sample[0]**.

Passing Single-Dimension Arrays to Functions

In C/C++, you cannot pass an entire array as an argument to a function. You can, however, pass to the function a pointer to an array by specifying the array's name without an index.

int main(void)

{

int i[10];

func1(i);

.

.

}

If a function receives a single-dimension array, you may declare its formal parameter in one of three ways: as a pointer, as a sized array, or as an unsized array.

void func1(int \*x) /\* pointer \*/

void func1(int x[10]) /\* sized array \*/

void func1(int x[]) /\* unsized array \*/

The length of the array doesn't matter as far as the function is concerned because C/C++ performs no bounds checking.

void func1(int x[32]) also works because the compiler generates code that instructs **func1( )** to receive a pointer—it does not actually create a 32-element array.

Null-Terminated Strings

By far the most common use of the one-dimensional array is as a character string. C++ supports two types of strings. The first is the *null-terminated string*, which is a null-terminated character array. (A null is zero.) Thus a null-terminated string contains the characters that comprise the string followed by a null. This is the only type of string defined by C, and it is still the most widely used. Sometimes null-terminated strings are called *C-strings*. C++ also defines a string class, called **string**, which provides an object-oriented approach to string handling.

When declaring a character array that will hold a null-terminated string, you need to declare it to be one character longer than the largest string that it is to hold. For example, to declare an array **str** that can hold a 10-character string, you would write

char str[11];

This makes room for the null at the end of the string.

When you use a quoted string constant in your program, you are also creating a null-terminated string. A *string constant* is a list of characters enclosed in double quotes. For example,

"hello there"

You do not need to add the null to the end of string constants manually—the compiler does this for you automatically.

C/C++ supports a wide range of functions that manipulate null-terminated strings. The most common are

Name Function

strcpy(*s1*, *s2*) Copies *s2* into *s1*.

strcat(*s1*, *s2*) Concatenates *s2* onto the end of *s1*.

strlen(*s1*) Returns the length of *s1*.

strcmp(*s1*, *s2*) Returns 0 if *s1* and *s2* are the same; less than 0 if *s1*<*s2*; greater than 0 if *s1*>*s2*.

strchr(*s1*, *ch*) Returns a pointer to the first occurrence of *ch* in *s1*.

strstr(*s1*, *s2*) Returns a pointer to the first occurrence of *s2* in *s1*

These functions use the standard header file **string.h**. (C++ programs can also use the C++-style header **<cstring>**.)

Although C++ defines a string class, null-terminated strings are still widely used in existing programs. They will probably stay in wide use because they offer a high level of efficiency and afford the programmer detailed control of string operations.

Two-Dimensional Arrays

int d[10][20];

Two-dimensional arrays are stored in a row-column matrix, where the first index indicates the row and the second indicates the column.

In the case of a two-dimensional array, the following formula yields the number of bytes of memory needed to hold it:

bytes = size of 1st index x size of 2nd index x sizeof(base type)

Therefore, assuming 4-byte integers, an integer array with dimensions 10,5 would have

10 x 5 x 4 or 200 bytes allocated.

When a two-dimensional array is used as an argument to a function, only a pointer to the first element is actually passed. However, the parameter receiving a two-dimensional array must define at least the size of the rightmost dimension. (You can specify the left dimension if you like, but it is not necessary.) The rightmost dimension is needed because the compiler must know the length of each row if it is to index the array correctly. For example, a function that receives a two-dimensional integer array with dimensions 10,10 is declared like this:

void func1(int x[][10])

The compiler needs to know the size of the right dimension in order to correctly execute expressions such as

x[2][4]

inside the function. If the length of the rows is not known, the compiler cannot determine where the third row begins.

Arrays of Strings

To create an array of null-terminated strings, use a two-dimensional character array. The size of the left index determines the number of strings and the size of the right index specifies the maximum length of each string. The following code declares an array of 30 strings, each with a maximum length of 79 characters, plus the null terminator.

char str\_array[30][80];

Multidimensional Arrays

C/C++ allows arrays of more than two dimensions. The exact limit, if any, is determined by your compiler. The general form of a multidimensional array declaration is

*type name*[*Size1*][*Size2*][*Size3*]. . .[*SizeN*];

Arrays of more than three dimensions are not often used because of the amount of memory they require. For example, a four-dimensional character array with dimensions 10,6,9,4 requires

10 \* 6 \* 9 \* 4

or 2,160 bytes.

In multidimensional arrays, it takes the computer time to compute each index. This means that accessing an element in a multidimensional array can be slower than accessing an element in a single-dimension array.

When passing multidimensional arrays into functions, you must declare all but the leftmost dimension. For example, if you declare array **m** as

int m[4][3][6][5];

a function, **func1( )**, that receives **m**, would look like this:

void func1(int d[][3][6][5])

Of course, you can include the first dimension if you like.

Indexing Pointers

an array name without an index is a pointer to the first element in the array.

char p[10];

The following statements are identical:

p

&p[0]

Put another way,

p == &p[0]

evaluates to true because the address of the first element of an array is the same as the address of the array.

int \*p, i[10];

p = i;

p[5] = 100; /\* assign using index \*/

\*(p+5) = 100; /\* assign using pointer arithmetic \*/

This same concept also applies to arrays of two or more dimensions. For example, assuming that **a** is a 10-by-10 integer array, these two statements are equivalent:

a

&a[0][0]

Furthermore, the 0,4 element of **a** may be referenced two ways: either by array indexing, **a[0][4]**, or by the pointer, **\*((int \*)a+4)**. Similarly, element 1,2 is either **a[1][2]** or **\*((int \*)a+12)**. In general, for any two-dimensional array

a[j][k] is equivalent to \*((*base-type* \*)a+(j\**row length*)+k)

The cast of the pointer to the array into a pointer of its base type is necessary in order for the pointer arithmetic to operate properly. Pointers are sometimes used to access arrays because pointer arithmetic is often faster than array indexing.

Array Initialization

C/C++ allows the initialization of arrays at the time of their declaration.

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

char *array\_name*[*size*] = "*string*";

char str[11] = "I like C++";

This is the same as writing

char str[11] = {'I', ' ', 'l', 'i', 'k', 'e',' ', 'C',

'+', '+', '\0'};

int sqrs[2][2] = {

1, 1,

2, 4

}

When initializing a multidimensional array, you may add braces around the initializers for each dimension. This is called *subaggregate grouping*. For example, here is another way to write the preceding declaration.

int sqrs[2][2] = {

{1, 1},

{2, 4}

}

When using subaggregate grouping, if you don't supply enough initializers for a given group, the remaining members will be set to zero automatically.

Unsized Array Initializations

char e1[] = "Read error\n";

char e2[] = "Write error\n";

char e3[] = "Cannot open file\n";

Unsized array initializations are not restricted to one-dimensional arrays. For multidimensional arrays, you must specify all but the leftmost dimension.

int sqrs[][2] = {

{1, 1},

{2, 4}

}

Pointers

A *pointer* is a variable that holds a memory address. This address is the location of another object (typically another variable) in memory.

Pointer Variables

If a variable is going to hold a pointer, it must be declared as such. A pointer declaration consists of a base type, an \*, and the variable name.

*type \*name;*

The Pointer Operators

There are two special pointer operators: **\*** and **&**. The **&** is a unary operator that returns the memory address of its operand. (Remember, a unary operator only requires one operand.) For example,

m = &count;

The second pointer operator, **\***, is the complement of **&**. It is a unary operator that returns the value located at the address that follows. For example, if **m** contains the memory address of the variable **count**,

q = \*m;

Both **&** and **\*** have a higher precedence than all other arithmetic operators except the unary minus, with which they are equal.

Pointer Arithmetic

There are only two arithmetic operations that you may use on pointers: addition and subtraction.

Pointer Comparisons

You can compare two pointers in a relational expression. For instance, given two pointers **p** and **q**, the following statement is perfectly valid:

if(p<q) printf("p points to lower memory than q\n");

Arrays of Pointers

Pointers may be arrayed like any other data type. The declaration for an **int** pointer

array of size 10 is

int \*x[10];

To assign the address of an integer variable called **var** to the third element of the

pointer array, write

x[2] = &var;

To find the value of **var**, write

\*x[2]

If you want to pass an array of pointers into a function, you can use the same method that you use to pass other arrays—simply call the function with the array name without any indexes. For example, a function that can receive array **x** looks like this:

void display\_array(int \*q[])

Multiple Indirection

You can have a pointer point to another pointer that points to the target value. This situation is called *multiple indirection*, or *pointers to pointers*.

float \*\*newbalance;

Pointers to Functions

Even though a function is not a variable, it still has a physical location in memory that can be assigned to a pointer. This address is the entry point of the function and it is the address used when the function is called.

You obtain the address of a function by using the function's name without any parentheses or arguments.

#include <stdio.h>

#include <string.h>

void check(char \*a, char \*b,

int (\*cmp)(const char \*, const char \*));

int main(void)

{

char s1[80], s2[80];

int (\*p)(const char \*, const char \*);

p = strcmp;

gets(s1);

gets(s2);

check(s1, s2, p);

return 0;

}

void check(char \*a, char \*b,

int (\*cmp)(const char \*, const char \*))

{

printf("Testing for equality.\n");

if(!(\*cmp)(a, b)) printf("Equal");

else printf("Not Equal");

}

**Output**

****

C's Dynamic Allocation Functions

*Dynamic allocation* is the means by which a program can obtain memory while it is running. As you know, global variables are allocated storage at compile time. Local variables use the stack. However, neither global nor local variables can be added during program execution. Yet there will be times when the storage needs of a program cannot be known ahead of time. For example, a program might use a dynamic data structure, such as a linked list or binary tree. Such structures are inherently dynamic in nature, growing or shrinking as needed. To implement such a data structure requires that a program be able to allocate and free memory.

C++ actually supports two complete dynamic allocation systems: the one defined by C and the one specific to C++.

Memory allocated by C's dynamic allocation functions is obtained from the *heap—*the region of free memory that lies between your program and its permanent storage area and the stack. Although the size of the heap is unknown, it generally contains a fairly large amount of free memory.

The core of C's allocation system consists of the functions **malloc( )** and **free( )**.

Any program that uses these functions should include the header file **stdlib.h**. (A C++ program may also use the C++-style header **<cstdlib>**.) The **malloc( )** function has this prototype:

void \*malloc(size\_t *number\_of\_bytes*);

If there is not enough available memory to satisfy the **malloc( )** request, an allocation failure occurs and **malloc( )** returns a null.

char \*p;

p = malloc(1000); /\* get 1000 bytes \*/

Notice that no type cast is used to assign the return value of **malloc( )** to **p.** In C, a **void \*** pointer is automatically converted to the type of the pointer on the left side of an assignment. However, it is important to understand that this automatic conversion *does not* occur in C++. In C++, an explicit type cast is needed when a **void \*** pointer is assigned to another type of pointer. Thus, in C++, the preceding assignment must be written like this:

p = (char \*) malloc(1000);

The **free( )** function is the opposite of **malloc( )** in that it returns previously allocated memory to the system. The function **free( )** has this prototype:

void free(void \**p*);

Here, *p* is a pointer to memory that was previously allocated using **malloc( )**. It is critical that you *never* call **free( )** with an invalid argument; otherwise, you will destroy the free list.

Functions

A function may return any type of data except an array.

Call by Value, Call by Reference

The first is known as *call by value*. This method copies the *value of* an argument into the formal parameter of the subroutine. In this case, changes made to the parameter have no effect on the argument.

*Call by reference* is the second way of passing arguments to a subroutine. In this method, the *address* of an argument is copied into the parameter. Inside the subroutine, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C/C++ uses call by value to pass arguments.

Calling Functions with Arrays

When an array is used as a function argument, its address is passed to a function. This is an exception to the call-by-value parameter passing convention.

argc and argv—Arguments to main( )

The **argc** parameter holds the number of arguments on the command line and is an integer. It is always at least 1 because the name of the program qualifies as the first argument. The **argv** parameter is a pointer to an array of character pointers.

Normally, you use **argc** and **argv** to get initial commands into your program. In theory, you can have up to 32,767 arguments, but most operating systems do not allow more than a few.

What Does main( ) Return?

The **main( )** function returns an integer to the calling process, which is generally the operating system. Returning a value from **main( )** is the equivalent of calling **exit( )** with the same value. If **main( )** does not explicitly return a value, the value passed to the calling process is technically undefined. In practice, most C/C++ compilers automatically return 0, but do not rely on this if portability is a concern.

Recursion

When a function calls itself, a new set of local variables and parameters are allocated storage on the stack, and the function code is executed from the top with these new variables. A recursive call does not make a new copy of the function. Only the values being operated upon are new. As each recursive call returns, the old local variables and parameters are removed from the stack and execution resumes at the point of the function call inside the function. Recursive functions could be said to "telescope" out

and back.

Often, recursive routines do not significantly reduce code size or improve memory utilization over their iterative counterparts. Also, the recursive versions of most routines may execute a bit slower than their iterative equivalents because of the overhead of the repeated function calls. In fact, many recursive calls to a function could cause a stack overrun. Because storage for function parameters and local variables is on the stack and each new call creates a new copy of these variables, the stack could be exhausted.

The main advantage to recursive functions is that you can use them to create clearer and simpler versions of several algorithms. For example, the Quicksort algorithm is difficult to implement in an iterative way. Also, some problems, especially ones related to artificial intelligence, lend themselves to recursive solutions. Finally, some people seem to think recursively more easily than iteratively.

Function Prototypes

In C++ all functions must be declared before they are used. This is normally accomplished using a *function prototype*.

The general form of a function prototype is

*type func\_name(type parm\_name1, type parm\_name2,. . .,*

*type parm\_nameN);*

The use of parameter names is optional. However, they enable the compiler to identify any type mismatches by name when an error occurs, so it is a good idea to include them.

Not clear

**void sqr\_it(int\* i); /\* prototype \*/**

**int main(void)**

**{**

**int x;**

**x = 10;**

**sqr\_it(x); /\* type mismatch \*/**

**return 0;**

**}**

**void sqr\_it(int \*i)**

**{**

**\*i = \*i \* \*i;**

**}**

**--------------------Configuration: testing - Win32 Debug--------------------**

**Compiling...**

**testing.cpp**

**d:\Faltu\testing\testing.cpp(14) : error C2664: 'sqr\_it' : cannot convert parameter 1 from 'int' to 'int \*'**

**Conversion from integral type to pointer type requires reinterpret\_cast, C-style cast or function-style cast**

**Error executing cl.exe.**

**testing.exe - 1 error(s), 0 warning(s)**

Structures

A structure is a collection of variables referenced under one name, providing a convenient means of keeping related information together.

struct *struct-type-name* {

*type member-name*;

*type member-name*;

*type member-name*;

..

} *structure-variables*;

Bit-Fields

Unlike some other computer languages, C/C++ has a built-in feature called a *bit-field* that allows you to access a single bit. Bit-fields can be useful for a number of reasons, such as:

■ If storage is limited, you can store several Boolean (true/false) variables in one byte.

■ Certain devices transmit status information encoded into one or more bits within a byte.

■ Certain encryption routines need to access the bits within a byte.

To access individual bits, C/C++ uses a method based on the structure. In fact, a bit-field is really just a special type of structure member that defines how long, in bits, the field is to be. The general form of a bit-field definition is

struct *struct-type-name* {

type *name1* : *length*;

*type name2* : *length*;

...

*type nameN* : *length*;

} *variable*\_*list*;

struct status\_type {

unsigned delta\_cts: 1;

unsigned delta\_dsr: 1;

unsigned tr\_edge: 1;

unsigned delta\_rec: 1;

unsigned cts: 1;

unsigned dsr: 1;

unsigned ring: 1;

unsigned rec\_line: 1;

} status;

status = get\_port\_status();

if(status.cts) printf("clear to send");

if(status.dsr) printf("data ready");

You do not have to name each bit-field. This makes it easy to reach the bit you want, bypassing unused ones. For example, if you only care about the **cts** and **dsr** bits, you could declare the **status\_type** structure like this:

struct status\_type {

unsigned : 4;

unsigned cts: 1;

unsigned dsr: 1;

} status;

It is valid to mix normal structure members with bit-fields. For example,

struct emp {

struct addr address;

float pay;

unsigned lay\_off: 1; /\* lay off or active \*/

unsigned hourly: 1; /\* hourly pay or wage \*/

unsigned deductions: 3; /\* IRS deductions \*/

};

Unions

A *union* is a memory location that is shared by two or more different types of variables.

union *union-type-name* {

*type member-name;*

*type member-name*;

*type member-name;*

...

} *union-variables;*

For example:

union u\_type {

int i;

char ch;

};

Enumerations

An *enumeration* is a set of named integer constants that specify all the legal values

a variable of that type may have.

enum *enum-type-name* { *enumeration list* } *variable\_list*;

enum coin { penny, nickel, dime, quarter,

half\_dollar, dollar};

Using sizeof to Ensure Portability

The **sizeof** operator computes the size of any variable or type and can help eliminate machine-dependent code from your programs. This operator is especially useful where structures or unions are concerned.

Since **sizeof** is a compile-time operator, all the information necessary to compute the size of any variable is known at compile time. This is especially meaningful for **union**s, because the size of a **union** is always equal to the size of its largest member.

typedef

You can define new data type names by using the keyword **typedef**. You are not actually *creating* a new data type, but rather defining a new name for an existing type.

typedef *type newname*;

typedef float balance;

balance over\_due;

Reading and Writing Characters

int getchar(void);

int putchar(int *c*);

The **getchar( )** function waits until a key is pressed and then returns its value. The key pressed is also automatically echoed to the screen. The **putchar( )** function writes a character to the screen at the current cursor position.

As its prototype shows, the **getchar( )** function is declared as returning an integer. However, you can assign this value to a **char** variable, as is usually done, because the character is contained in the low-order byte. (The high-order byte is normally zero.) **getchar( )** returns **EOF** if an error occurs.

In the case of **putchar( ),** even though it is declared as taking an integer parameter, you will generally call it using a character argument. Only the low-order byte of its parameter is actually output to the screen. The **putchar( )** function returns the character written, or **EOF** if an error occurs. (The **EOF** macro is defined in **stdio.h** and is generally equal to −1.)

A Problem with getchar( )

There are some potential problems with **getchar( )**. Normally, **getchar( )** is implemented in such a way that it buffers input until ENTER is pressed. This is called *line*-*buffered* input; you have to press ENTER before anything you typed is actually sent to your program. Also, since **getchar( )** inputs only one character each time it is called, line-buffering may leave one or more characters waiting in the input queue, which is annoying in interactive environments. Even though Standard C/C++ specify that **getchar( )** can be implemented as an interactive function, it seldom is. Therefore, if the preceding program did not behave as you expected

Alternatives to getchar( )

int getch(void);

int getche(void);

The **getch( )** function waits for a keypress, after which it returns immediately. It does not echo the character to the screen. The **getche( )** function is the same as **getch( )**, but the key is echoed.

Reading and Writing Strings

**gets( )** and **puts( )** enable you to read and write strings of characters.

char \*gets(char \**str*);

The **gets( )** function reads a string of characters entered at the keyboard and places them at the address pointed to by its argument. You may type characters at the keyboard until you press ENTER. The carriage return does not become part of the string; instead, a null terminator is placed at the end and **gets( )** returns.

You need to be careful when using **gets( )** because it performs no boundary checks on the array that is receiving input.

int puts(const char \**str*);

The **puts( )** function writes its string argument to the screen followed by a newline.

A call

to **puts( )** requires far less overhead than the same call to **printf( )** because **puts( )** can only output a string of characters⎯it cannot output numbers or do format conversions. Therefore, **puts( )** takes up less space and runs faster than **printf( )**.

Formatted Console I/O

The functions **printf( )** and **scanf( )** perform formatted output and input⎯that is, they can read and write data in various formats that are under your control. The **printf( )** function writes data to the console. The **scanf( )** function, its complement, reads data from the keyboard.

int printf(const char \**control*\_*string*, ...);

The **printf( )** function returns the number of characters written or a negative value if an error occurs.

The **printf( )** function accepts a wide variety of format specifiers, as shown in

|  |  |
| --- | --- |
| **Code** | **Format** |
| **%c** | Character |
| **%d** | Signed decimal integers |
| **%i** | Signed decimal integers |
| **%e** | Scientific notation (lowercase e) |
| **%E** | Scientific notation (uppercase E) |
| **%f** | Decimal floating point |
| **%g** | Uses %e or %f, whichever is shorter |
| **%G** | Uses %E or %F, whichever is shorter |
| **%o** | Unsigned octal |
| **%s** | String of characters |
| **%u** | Unsigned decimal integers |
| **%x** | Unsigned hexadecimal (lowercase letters) |
| **%X** | Unsigned hexadecimal (uppercase letters) |
| **%p** | Displays a pointer |
| **%n** | The associated argument must be a pointer to an integer. This specifier causes the number of characters written so far to be put into that integer. |
| **%%** | Prints a % sign |

**<Need to do chapter 8 again>**

**<Need to do chapter 9 >**

The Preprocessor