UNIT-I
OBJECT ORIENTED THINKING

DIFFERENT PARADIGMS FOR PROBLEM SOLVING

Paradigm definition

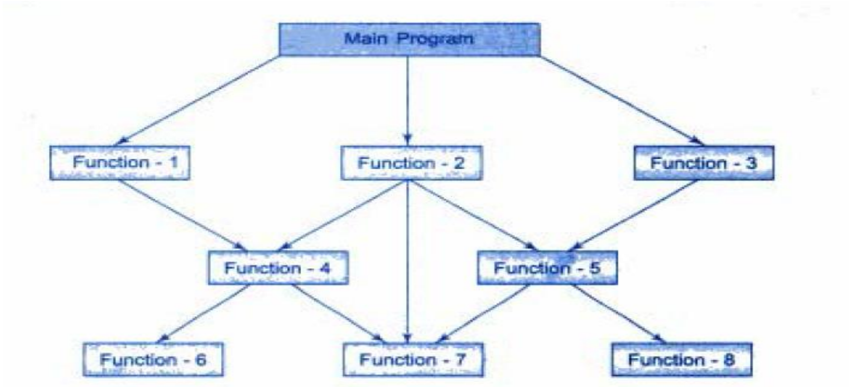
A paradigm is a way in which a computer language looks at the problem to be solved.

Evolution of Paradigms

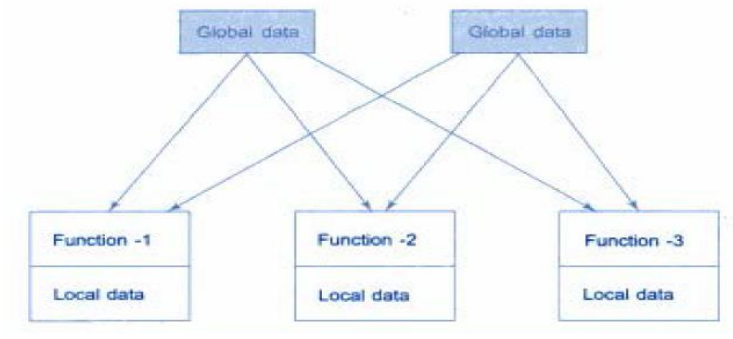
Since the invention of computers, many programming approaches have been developed. The primary motivation of doing so is to handle the increasing complexity of programs and to make them reliable and maintainable.

The following are the different paradigms for problem solving,

1. **Monolithic programming**
    - This is the main technique used in 1980's.
    - The program is written with a single function. A program is not divided into parts i.e. statements are written in sequence.
    - When the program size increases it failed to show the desired result in terms of bug free, easy to maintain and reusable programs.
    - The concept of sub programs does not exist, and hence is useful for small programs.
2. **Procedure Oriented Programming**
    - It basically consists of writing a list of instructions for the computer to follow and break down the code and organize these instructions into manageable segments or groups known as functions.
    - In this, the problem is viewed as a sequence of things to be done such as reading, calculating and printing. A number of functions are written to accomplish these tasks.



    - In a multi function program, many important data items are placed as global so that they may be accessed by all the functions. Each function may have its local data.



Drawbacks

- Global data are more vulnerable to an inadvertent change by a function.

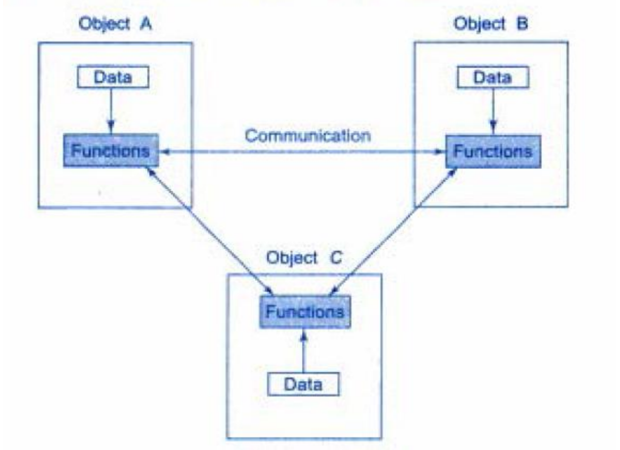
- In a large program it is very difficult to identify what data is used by which function. This provides an opportunity for bugs to creep in.
- It does not model real world problems very well. This is because functions are action oriented and do not really corresponding to the elements of the problem.

Characteristics

- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs *top·down* approach in program design.

3. Object Oriented Programming

- This is the most recent concept among programming paradigms.
- It is an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as template for creating copies of such modules on demand.
- The major motivating factor in the invention of object oriented approach is *to* remove some of the flaws encountered in the procedural approach.



Features

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- *Follows bottom-up* approach in program design.

NEED FOR OBJECT-ORIENTED PARADIGM

- The object oriented programming paradigm is a methodology for producing re usable software components.
- It promotes efficient design and development of software systems using reusable components that can be quickly and safely assembled into larger systems.
- It produces reusable code/objects because of encapsulation and inheritance.

- The data is protected because it can be altered only by the encapsulated methods.
- It is more efficient to write programs which use pre-defined objects.
- The storage structure and/or procedures within an object type could be altered if required without affecting programs that make use of that object type.
- New functions can easily be added to objects by using inheritance
- The code produced is likely to contain fewer errors because pretested objects are being used.
- Less maintenance effort will be required by the developer because objects can be reused.

DIFFERENCE BETWEEN OBJECT ORIENTED PROGRAMMING AND PROCEDURE ORIENTED PROGRAMMING

S.NO	Object oriented Programming	Procedure Oriented Programming
1	Emphasis is on data	Emphasis is on doing things
2	Programs are divided into Objects	Programs are divided into Functions
3	Employs Botton up approach	Employs Top down approach
4	Modification potential is high	Modification potential is low
5	Data is hidden and cannot be accessed by external functions	Data is open and can be accessed by any functions
6	Suitable for solving big problems	Not Suitable for solving big problems
7	It needs more memory than POP	It needs less memory
8	Supports Polymorphism, Inheritance, abstraction and Encapsulation	Does not supports Polymorphism, Inheritance, abstraction and Encapsulation
9	Example languages are C++, Java	Example languages are C,VB,FORTRAN, COBOL

OVERVIEW OF OOP CONCEPTS

1. **Abstraction**
   - *Abstraction* refers to the act of representing essential features without including the background details or explanations.
   - Classes use the concept of abstraction and are defined as a list of abstract *attributes* such as size, weight and cost and *functions* to operate on these attributes.
   - They encapsulate all the essential properties of the objects that are to be created.
   - The attributes are sometimes called *data numbers* because they hold information.
   - The functions that operate on these data are sometimes called *methods or member functions*.
   - Since the classes use the concept of data abstraction, they are known as *Abstract Data Types* (ADT).

2. **Encapsulation**

- *Encapsulation* is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.
- In an object-oriented language, code and data may be combined in such a way that a self-contained "black box" is created.
- When code and data are linked together in this fashion, an *object* is created. In other words, an object is the device that supports encapsulation.
- Within an object, code, data, or both may be *private* to that object or *public*.
- Private code or data is known to and accessible only by another part of the object. That is, private code or data may not be accessed by a piece of the program that exists outside the object.
- When code or data is public, other parts of your program may access it even though it is defined within an object. Typically, the public parts of an object are used to provide a controlled interface to the private elements of the object.

3. Polymorphism

- Object-oriented programming languages support *polymorphism*, which is characterized by the phrase "one interface, multiple methods."
- In simple terms, polymorphism is the attribute that allows one interface to control access to a general class of actions. The specific action selected is determined by the exact nature of the situation.
- For example, you might have a program that defines three different types of stacks. One stack is used for integer values, one for character values, and one for floating-point values. Because of polymorphism, you can define one set of names, **push( )** and **pop( )**, that can be used for all three stacks.
- Polymorphism helps reduce complexity by allowing the same interface to be used to access a general class of actions. It is the compiler's job to select the *specific action* (i.e., method) as it applies to each situation. You, the programmer, don't need to do this selection manually. You need only remember and utilize the *general interface*.

4. Inheritance

- *Inheritance* is the process by which one object can acquire the properties of another object.
- This is important because it supports the concept of *classification*. If you think about it, most knowledge is made manageable by hierarchical classifications.
- For example, a Red Delicious apple is part of the classification *apple*, which in turn is part of the *fruit* class, which is under the larger class *food*. Without the use of classifications, each object would have to define explicitly all of its characteristics. However, through the use of classifications, an object need only define those qualities that make it unique within its class. It is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.

C++ BASICS

Origin of C++

C++ began as an expanded version of C. The C++ extension was invented by **Bjarne Stroupstrup in 1979** at **Bell laboratories** He initially called the new language as "C **with Classes**" but renamed it as "**C++**" in 1983.

STRUCTURE OF A C++ PROGRAM

Most C++ programs has the following general form,

```
#include
base class declarations
derived class declarations
non-member function prototypes
int main()
{

        //.....
}
non member function definition
```

Sample C++ Program

```
#include<iostream.h>
int main()
{
        int i;
        cout<< “enter a number”:
        cin>>i;
        cout<<i<<” squared is“<<i*i<<endl;
        return 0;
}
```

Output:

```
enter a number 10
10 squared is 100
```

Header File

Header <iostream> is included. This header supports C++ style of I/O operations.

Input Operator

cin>>i;

This is an input statement and causes the program to wait for the user to type in a number.

Operator >> is known as **extraction or getfrom** operator. It takes the value from the keyboard and assigns it to the variable on its right. Similar to scanf() inn C.

Output Operator

cout<< “enter a number”:

This causes the string in quotation marks to be displayed on the screen.

Operator << is called **insertion or putto** operator. It inserts (or sends) the contents of the variable on its right to the object on its left. Similar to print() in C.

DATA TYPES

There are 7 different data types in C++.They are

1. Character(char)
2. Integer(int)
3. Floating-point(float)
4. Double floating-point(double)
5. Valueless(void)
6. Boolean(bool)

7. Wide Character(Wchar_t)

6. bool Data Type

C++ defines a built-in Boolean type called **bool**. Objects of type **bool** can store only the values **true** or **false**, which are keywords defined by C++. Automatic conversions take place which allow **bool** values to be converted to integers, and vice versa. Specifically, any non-zero value is converted to **true** and zero is converted to **false**. The reverse also occurs; **true** is converted to 1 and **false** is converted to zero.

General form: bool b1=true;

7.Wide Characters

C++ define wide characters which are 16 bits long. To specify a wide character precede the character with an **L**.

General form: wchar_t wc;
wc = L'A';

Here, **wc** is assigned the wide-character constant equivalent of A. The type of wide characters is **wchar_t**. In C, this type is defined in a header file and is not a built-in type. In C++, **wchar_t** is built in.

Program 1: Write a C++ program to demonstrate different data types available in C++

```
#include <iostream>
using namespace std;

int main ()
{
  bool b = true;
  wchar_t w = L'A';
  int i;
  char ch;
  float fl;
  double d1;

  cout << "Enter a character: ";
  cin >> ch;
  cout << "\nYou entered: " << ch;

  cout << "\n\nEnter a floating-point number: ";
  cin >> fl;
  cout << "\nYou entered: " << fl;

  cout << "\n Enter a integer";
  cin >> i;
  cout << "\n You entered :" << i;

  cout << "\n Enter a double";
  cin >> d1;
  cout << "\n You entered :" << d1;

  cout << "\n boolean value is :" << b;
  cout << "\n Wide character value::" << w << '\n';

  return 0;
}
```

Output:

```
Enter a character: a

You entered: a

Enter a floating-
point number: 3345

You entered: 3345
 Enter a integer56

 You entered :56
 Enter a double57

 You entered :57
 boolean value is :1
 Wide character value::65
```

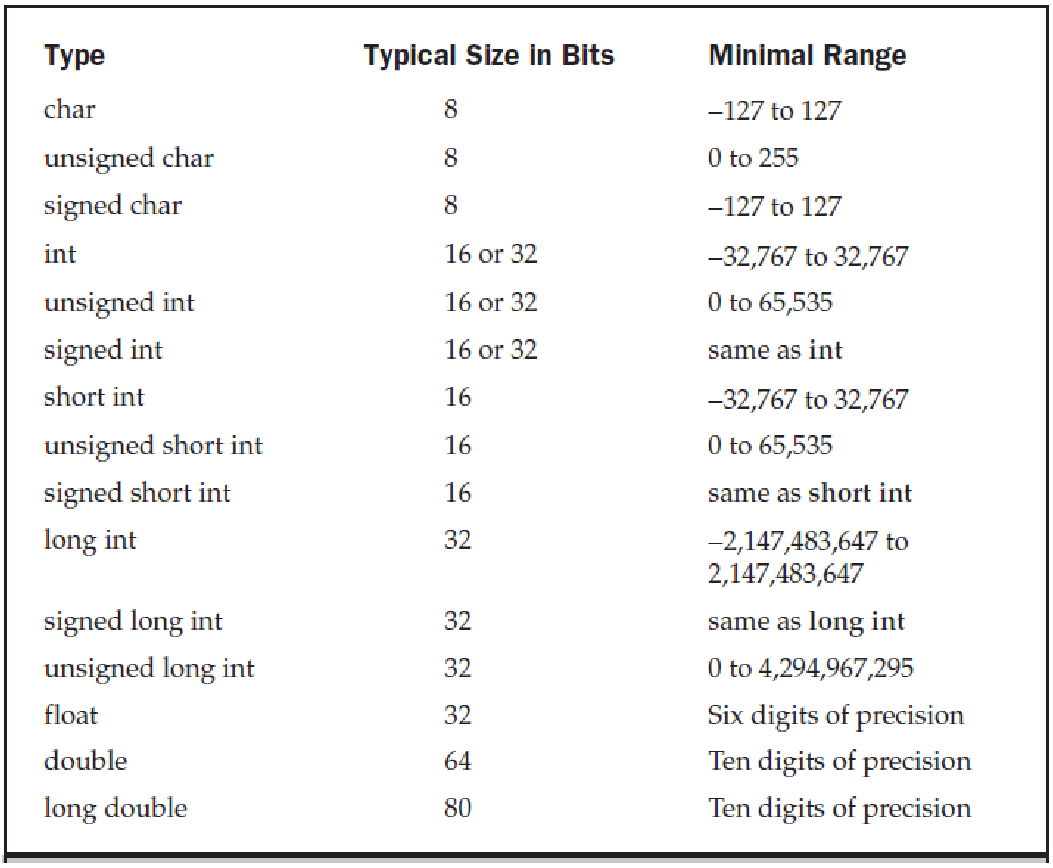
Program 2: Write a C++ program to know sizes of different data types available in C++

```
#include <iostream>
using namespace std;

int main()
{
        cout << "Size of char : " << sizeof(char) << " byte" << endl;

         cout << "Size of int : " << sizeof(int) << " bytes" << endl;

        cout << "Size of short int : " << sizeof(short int)  << " bytes" << endl;

        cout << "Size of long int : " << sizeof(long int)   << " bytes" << endl;

        cout << "Size of signed long int : " << sizeof(signed long int)  << " bytes" << endl;

        cout << "Size of unsigned long int : "<< sizeof(unsigned long int)<< " bytes" << endl;

        cout << "Size of float : " << sizeof(float)   << " bytes" <<endl;

        cout << "Size of double : " << sizeof(double)   << " bytes" << endl;

        cout << "Size of wchar_t : " << sizeof(wchar_t)  << " bytes" <<endl;

        return 0;
}
```

Output:

```
Size of char : 1 byte
Size of int : 4 bytes
Size of short int : 2 bytes
Size of long int : 8 bytes
Size of signed long int : 8 bytes
Size of unsigned long int : 8 bytes
Size of float : 4 bytes
Size of double : 8 bytes
Size of wchar_t : 4 bytes
```

Data types size and Range



Type	Typical Size in Bits	Minimal Range
char	8	–127 to 127
unsigned char	8	0 to 255
signed char	8	–127 to 127
int	16 or 32	–32,767 to 32,767
unsigned int	16 or 32	0 to 65,535
signed int	16 or 32	same as int
short int	16	–32,767 to 32,767
unsigned short int	16	0 to 65,535
signed short int	16	same as short int
long int	32	–2,147,483,647 to 2,147,483,647
signed long int	32	same as long int
unsigned long int	32	0 to 4,294,967,295
float	32	Six digits of precision
double	64	Ten digits of precision
long double	80	Ten digits of precision

Modifying the Basic Types

Except for type **void**, the basic data types may have various modifiers preceding them. You use a *modifier* to alter the meaning of the base type to fit various situations more precisely.

You can apply the modifiers **signed**, **short**, **long**, and **unsigned** to integer base types. You can apply **unsigned** and **signed** to characters. You may also apply **long** to **double**.

The list of modifiers is shown here:

1. signed
2. unsigned
3. long
4. short

VARIABLES

A *variable* is a named location in memory that is used to hold a value that may be modified by the program.

Declaration of a variable

All variables must be declared before they can be used.

General form: ***type variable_list;***

Here, *type* must be a valid data type plus any modifiers, and *variable_list* may consist of one or more identifier names separated by commas.

Examples:
```
int i,j,l;
short int si;
unsigned int ui;
double balance, profit, loss;
```

Initialization of a variable

We can assign a value to a variable.

General form: ***variable= expression;***

Example: i=10;

We can initialize a variable at the time of declaration.

General form: type variable= ***expression;***

Example: int i=10;

Where Variables Are Declared

Variables will be declared in three basic places:

1. Inside functions (local variables)
2. In the definition of function parameters(formal parameters)
3. And outside of all functions (global variables)

1. Local Variables

Variables that are declared inside a function are called *local variables*. Local variables may be referenced only by statements that are inside the block in which the variables are declared. In other words, local variables are not known outside their own code block.

Local variables exist only while the block of code in which they are declared is executing. That is, a local variable is created upon entry into its block and destroyed upon exit. The most common code block in which local variables are declared is the function.

For example, consider the following two functions:

```
void func1(void)
{
int x;
x = 10;
}


void func2(void)
{
int x;
x = -199;
}
```

The integer variable **x** is declared twice, once in **func1( )** and once in **func2( )**. The **x** in **func1( )** has no bearing on or relationship to the **x** in **func2( )**. This is because each **x** is known only to the code within the block in which it is declared.

Program 1: Write a C++ program to demonstrate Local Variables

```
#include <iostream>
using namespace std;

int main()
{
float f;
double d;

cout << "Enter two floating point numbers: ";
cin >> f >> d;

cout << "Enter a string: ";
char str[80]; // str declared here, just before 1st use
cin >> str;

cout <<"printing received values"<<endl<< f << " " << d << " " << str;
return 0;
}
```

Output:

```
Enter two floating point numbers: 9.9

17.21
Enter a string: keerthi
printing received values
9.9 17.21 keerthi
```

Important difference between C and C++

An important difference between C and C++ is when local variables can be declared. In C89, you must declare all local variables used within a block at the start of that block. You cannot declare a variable in a block after an "action" statement has occurred. For example, in C89, this fragment is incorrect:

```
/* Incorrect in C89. OK in C++. */
int f()
{
int i;
i = 10;
int j; /* won't compile as a C program */
j = i*2;
return j;
}
```

In a C89 program, this function is in error because the assignment intervenes between the declaration of **i** and that of **j**. However, when compiling it as a C++ program, this fragment is perfectly acceptable. In C++ (and C99) you may declare local variables at any point within a block—not just at the beginning.

2. 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. They behave like any other local variables inside the function. As shown in the following program fragment, their declarations occur after the function name and inside parentheses:

```
/* Return 1 if c is part of string s; 0 otherwise */
int is_in(char *s, char c)
{
while(*s)
if(*s==c) return 1;
else s++;
return 0;
}
```

The function **is_in( )** has two parameters: **s** and **c**. This function returns 1 if the character specified in **c** is contained within the string **s**; 0 if it is not.

3. Global Variables

Unlike local variables, *global variables* are known throughout the program and may be used by any piece of code. Also, they will hold their value throughout the program's execution. You create global variables by declaring them outside of any function. Any expression may access them, regardless of what block of code that expression is in.

Program 2: Write a C++ program to demonstrate Global Variables

```
#include <iostream>
using namespace std;

int count; /* count is global */
void func1(void);
void func2(void);
int main(void)
{
count = 100;
func1();
return 0;
}
void func1(void)
{
int temp;
temp = count;
cout<<"count is (from func1)"<< count; /* will print 100 */
func2();
}

void func2(void)
{
```

```
int temp;
temp = count;
cout<<"count is (from func2)"<< count; /* will print 100 */
}
```

Output:

```
count is (from func1)100count is (from func2)100
```

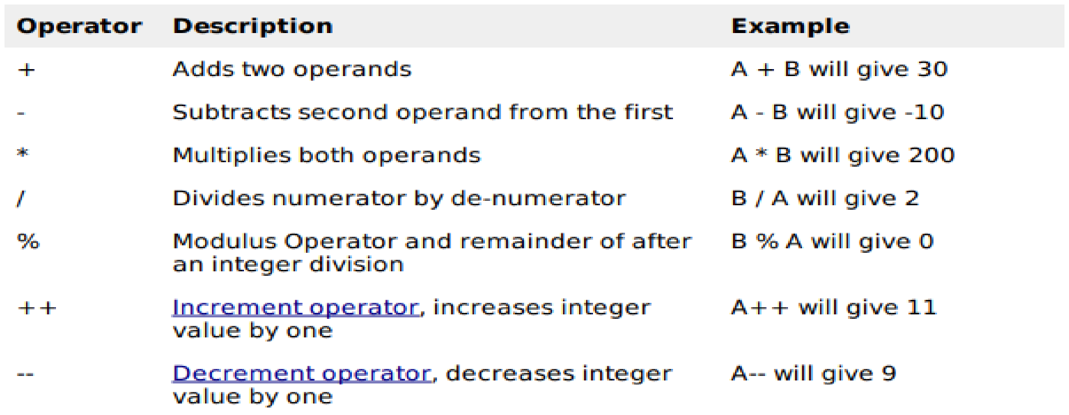
OPERATORS

C++ is rich in built-in operators. There are four main classes of operators: *arithmetic*, *relational*, *logical*, and *bitwise*. In addition, there are some special operators for particular tasks.

1. Arithmetic Operators

These are defined to perform basic arithmetic operations. The operators **+**, **−**, *****, and **/** work as they do in most other computer languages. You can apply them to almost any built-in data type.

Assume variable A holds 10 and variable B holds 20



Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator, increases integer value by one	A++ will give 11
--	Decrement operator, decreases integer value by one	A-- will give 9

Both the increment and decrement operators may either precede (prefix) or follow (postfix) the operand. For example,

```
x = x+1;
```

can be written

```
++x;
```

or

```
x++;
```

There is, however, a difference between the prefix and postfix forms when you use these operators in an expression. When an increment or decrement operator precedes its operand, the increment or decrement operation is performed before obtaining the value of the operand for use in the expression. If the operator follows its operand, the value of the operand is obtained before incrementing or decrementing it. For instance,

```
x = 10;
y = ++x;
```

sets **y** to 11. However, if you write the code as

```
x = 10;
y = x++;
```

y is set to 10. Either way, **x** is set to 11; the difference is in when it happens.

Precedence of the Arithmetic Operators

highest	++ – –
	– (unary minus)
	* / %
lowest	+ –

PROGRAM 3: ARITHEMATIC OPERATORS

```
#include <iostream>
#include<conio.h>
using namespace std;

int main()
{
	int num1, num2, res;

	cout<<"Enter any two number: ";
	cin>>num1>>num2;

	res = num1 + num2;
	cout<<"\n";
	cout<<num1<<" + "<<num2<<" = "<<res<<endl;

	res = num1 - num2;
	cout<<num1<<" - "<<num2<<" = "<<res<<endl;

	res = num1 * num2;
	cout<<num1<<" * "<<num2<<" = "<<res<<endl;

	res = num1 / num2;
	cout<<num1<<" / "<<num2<<" = "<<res<<endl;

	res = num1 % num2;
	cout<<num1<<" % "<<num2<<" = "<<res<<endl;

	getch();
}
```

OUTPUT:

```
Enter any two number: 2
3

2 + 3 = 5
2 - 3 = -1
2 * 3 = 6
2 / 3 = 0
2 % 3 = 2
```

2. Relational Operators

Relational operators refer to the relationships that values can have with one another. The result of relational operators is either true or false.

Assume variable A holds 10 and variable B holds 20

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	$A == B$ is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	$A! = B$ is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	$A > B$ is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	$A < B$ is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	$A >= B$ is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	$A <= B$ is true.

PROGRAM 4: RELATIONAL OPERATORS

```
#include <iostream>
#include<conio.h>
using namespace std;

int main()
{
        int p, q;
        int res;
        cout<<"Enter any two number: ";
        cin>>p>>q;

        cout<<"\n";
        cout<<"p    q    p<q    p<=q    p==q    p>q    p>=q    p!=q\n\n";

        res = p<q;
        cout<<p<<"    "<<q<<"    "<<res<<"      ";
        res = p<=q;
        cout<<res<<"       ";
        res = p==q;
        cout<<res<<"       ";
        res = p>q;
        cout<<res<<"      ";
        res = p>=q;
        cout<<res<<"       ";
        res = p!=q;
        cout<<res<<endl;

        getch();
}
```

OUTPUT:

```
Enter any two number: 3 4
p    q    p<q    p<=q    p==q    p>q    p>=q    p!=q
3    4    1      1       0       0      0       1
```

3. Logical Operators

Logical refers to the ways these relationships can be connected or combined. The result of logical operators is either true or false.

Assume variable A holds 1 and variable B holds 0

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	A && B is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	A || B is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!A && B is true.

The truth table for the logical operators is

p	q	p && q	p || q	!p
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

Precedence of the Relational and Logical operators:

Highest	!
	> >= < <=
	== !=
	&&
Lowest	||

PROGRAM 5: LOGICAL OPERATORS

```
#include <iostream>
#include<conio.h>
using namespace std;

int main()
{
        int res;

        res = (6 <= 6) || (5 <3);
        cout<<res<<endl;

        res = (6 <= 6) && (5 < 3);
        cout<<res<<endl;

        res = !(6 <= 6);
        cout<<res<<endl;

        res = !(5 > 9);
        cout<<res<<endl;

        getch();
}
```

OUTPUT:

```
1
0
0
1
```

4. Bitwise Operators

C++ supports a full complement of bitwise operators. *Bitwise operation* refers to testing, setting, or shifting the actual bits in a byte or word, which correspond to the **char** and **int** data types and variants. You cannot use bitwise operations on **float**, **double**, **long double**, **void**, **bool**, or other, more complex types.

Bitwise operations most often find application in device drivers—such as modem programs, disk file routines, and printer routines — because the bitwise operations can be used to mask off certain bits, such as parity.

Assume if A = 60; and B = 13; now in binary format they will be as follows:

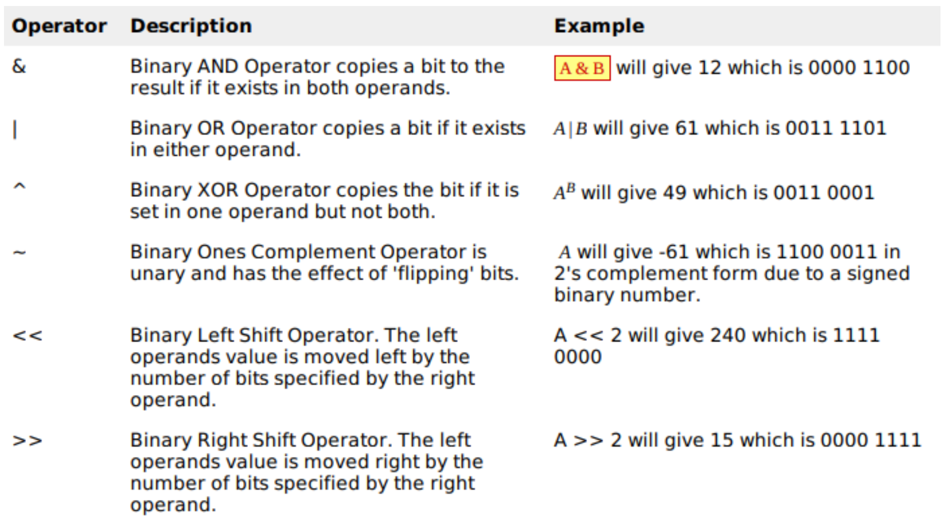
A = 0011 1100
B = 0000 1101

A&B = 0000 1100
A|B = 0011 1101
A^B = 0011 0001
~A = 1100 0011

You can combine several operations together into one expression, as shown here:

10>5 && !(10<9) || 3<=4

Assume variable A holds 60 and variable B holds 13



Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	A & B will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	A|B will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	A^B will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	A will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

PROGRAM 6: BITWISE OPERATORS

```
#include <iostream>
#include<conio.h>
using namespace std;

int main()
{
	unsigned int a = 60;	// 60 = 0011 1100
	unsigned int b = 13;	// 13 = 0000 1101
	int c = 0;

	c = a & b;		// 12 = 0000 1100
	cout<<"a = 0011 1100 (60)\tand\tb = 0000 1101 (13)\n\n";
	cout<<"a & b = "<<c<<endl;

	c = a | b;		// 61 = 0011 1101
	cout<<"a | b = "<<c<<endl;

	c = a ^ b;		// 49 = 0011 0001
	cout<<"a ^ b = "<<c<<endl;

	c = ~a;			// -61 = 1100 0011
	cout<<"~a = "<<c<<endl;

	getch();
}
```

OUTPUT:

```
a = 0011 1100 (60)      and     b = 0000 1101 (13)

a & b = 12
a | b = 61
a ^ b = 49
~a = -61
```

Special operators

a. The ? Operator

C++ contains a very powerful and convenient operator that replaces certain statements of the if-then-else form. The ternary operator **?** takes the general form

Exp1 ? Exp2 : Exp3;

where *Exp1*, *Exp2*, and *Exp3* are expressions.

The **?** operator works like this: *Exp1* is evaluated. If it is true, *Exp2* is evaluated and becomes the value of the expression. If *Exp1* is false, *Exp3* is evaluated and its value becomes the value of the expression.

For example,

```
	x = 10;
	y = x>9 ? 100 : 200;
```

y is assigned the value 100. If **x** had been less than 9, **y** would have received the value 200.

b. &(the address of) Pointer Operators

It is a unary operator that returns the memory address of its operand.

Example: m = &count;

places into **m** the memory address of the variable **count**.

c. *(at address) Pointer Operators

It is a unary operator that returns the value of the variable located at the address that follows it.

For example, if **m** contains the memory address of the variable **count**,

q = *m;

places the value of **count** into **q**. Now **q** has the value 100 because 100 is stored at location 2000, the memory address that was stored in **m**.

d. sizeof

sizeof is a unary compile-time operator that returns the length, in bytes, of the variable or parenthesized type-specifier that it precedes.

Example: sizeof(int) will display 4

e. The Comma Operator

The comma operator strings together several expressions. 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,

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

first assigns **y** the value 3 and then assigns **x** the value 4.

f. The Dot (.) and Arrow (>) Operators

The . (dot) and the >(arrow) operators access individual elements of structures and unions. In C++, the dot and arrow operators are also used to access the members of a class.

The dot operator is used when working with a structure or union directly. The arrow operator is used when a pointer to a structure or union is used.

```
struct employee
{
char name[80];
int age;
float wage;
} emp;
struct employee *p = &emp; /* address of emp into p */
```

you would write the following code to assign the value 123.23 to the **wage** member of structure variable **emp**:

```
emp.wage = 123.23;
```

However, the same assignment using a pointer to **emp** would be

```
p->wage = 123.23;
```

g. The [ ] and ( ) Operators

Parentheses are operators that increase the precedence of the operations inside them.

Square brackets perform array indexing

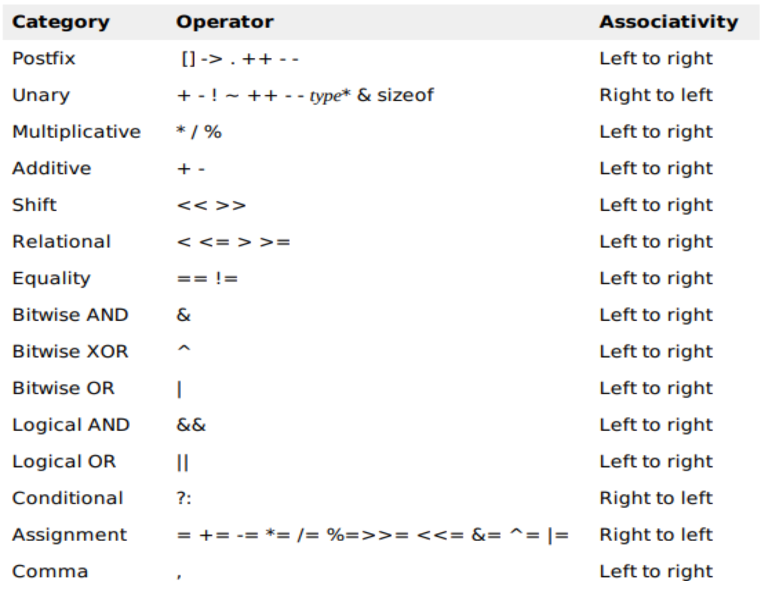
    s[3] = 'X';

h. The Assignment Operator

You can use the assignment operator within any valid expression. C++ uses a single equal sign to indicate assignment

General form : *variable_name = expression;*

Operators Precedence in C++



Category	Operator	Associativity
Postfix	[] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - *type** & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

EXPRESSIONS

An *expression* in C++ is any valid combination of Operators, constants, and variables. Expressions 1nay be of the following seven types:

- Constant expressions
- Integral expressions
- Float expressions
- Point.er expressions
- Relational expressions
- Logical expressions
- Bitwise expressions

Constant Expressions

Constant Expressions consist of only constant va1ues.

Examples: 15
20 + 5 / 2.0
'x'

Integral Expressions

Integral Expressions are those which produce integer results after implementing all the automatic and explicit type conversions.

Examples: m
m * n -5
m • 'x'

5 + int(2.0)

where m and n are integer variables.

Float Expressions

Float Expressions are those which, after all conversions, produce floating-point results.

Examples: x + y
x * y / 10
5 * float(10)
10.75

where x and y are floating-point variables.

Pointer Expressions

Pointer Expressions produce address values.

Examples: &m
ptr
ptr + l
“xyz”

where m is a variable and ptr is a pointer.

Relational Expressions

Relational Expressions yield results of type bool which takes a value true or false.

Examples: x<=y
a+b == c+d
m+-n > 100

When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results compared. Relational expressions are also known as ***Boolean expressions.***

Logical Expressions

Logical Expressions combine two or more relationa1 expressions and produces bool type results.

Examples: a>b && x0010
x==10 || y==5

Bitwise Expressions

Bitwise Expressions are used to manipulate data at bit level. They are basically used for testing or shifting bits.

Examples: x << 3 *// Shift three bit position to left*
y >>1 // *Shift* one *bit position to right*

Shift operators are often used for multiplication and division by powers of two.

ANSI C++ has introduced what ore termed as *operator keyword* that can be used as alternative representation for operator symbols.

Special Assignment Expressions

Chained Assignment

x=(y=10);
or
x=y=10;

First 10 is assigned to y and then to x.

A chained statement cannot be used to initialize variables at the time of declaration. For instance, the statement

```
float a =b =12.34;
```

is illegal. This may be written as

```
float a=12.34,b=12.34
```

Embedded Assignment

```
x= (y= 50) + 10 ;
```

(y = 50) is an assignment expression known as embedded assignment. Here, the value 50 is assigned to y and then the result 50+ 10 = 60 is assigned to x. This statement is identical to

```
y = 50;
x =y + 10;
```

Compound Assignment

Like C, C+-+ supports a *compound assignment operator* which is a combination of the assignment operator with a binary arithmetic operator. For example, the simple assignment statement

```
x= x + 10;
```

may be written as

```
x+= 10;
```

The operator += is known as *compound assignment operator* or *short-hand assignment operator*. The general form of the compound assignment operator is:

```
Variable1 op= variable2;
```

where *op* is a binary arithmetic operator. This means that
variable1 = variable op variable2;

ORDER OF EVALUATION OF EXPRESSIONS

C++ does not specify the order in which the sub expressions 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 sub expressions 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*.) Once this step has been completed, all other conversions are done operation by operation, as described in the following type conversion algorithm:

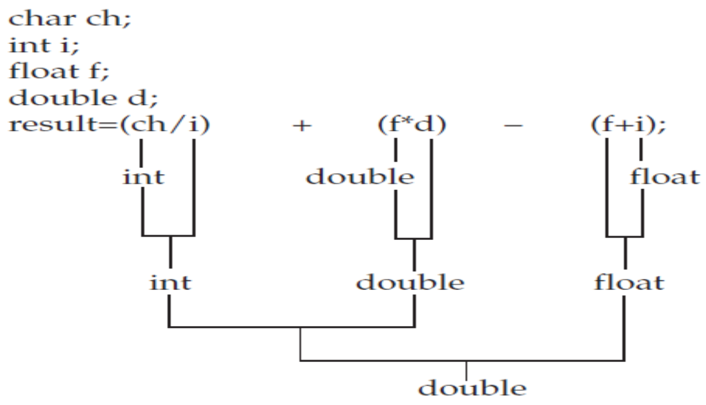
```
IF an operand is a long double
THEN the second is converted to long double
ELSE IF an operand is a double
THEN the second is converted to double
ELSE IF an operand is a float
THEN the second is converted to float
ELSE IF an operand is an unsigned long
```

THEN the second is converted to unsigned long
ELSE IF an operand is long
THEN the second is converted to long
ELSE IF an operand is unsigned int
THEN the second is converted to unsigned int

Additional special case: If one operand is long and the other is unsigned int, and if the value of the unsigned int cannot be represented by a long, both operands are converted to unsigned long.

Example:

First, the character ch is converted to an integer. Then the outcome of ch/i is converted to a double because f*d is double. The outcome of f+i is float, because f is a float. The final result is double.



```
char ch;
int i;
float f;
double d;
result=(ch/i)      +      (f*d)      –      (f+i);
```

Casts

You can force an expression to be of a specific type by using a *cast*.

General form : *(type) expression*

where *type* is a valid data type.

Example:

To make sure that the expression x/2 evaluates to type float, write (float) x/2

PROGRAM: TYPE CONVERSION

```
#include <iostream>
#include<conio.h>
using namespace std;

int main()
{
        float res;
        float f1=15.5, f2=2;

        res = (int)f1/(int)f2;
        cout<<res<<endl;

        res = (int)(f1/f2);
        cout<<res<<endl;

        res = f1/f2;
        cout<<res;
        getch();
}
```

OUTPUT:

```
7
7
7.75
```

FLOW CONTROL STATEMENTS

A) SELECTION STATEMENTS

C++ supports two types of selection statements:

a) **if**
b) **switch**.

if

The general form of the **if** statement is

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

where a *statement* may consist of a single statement, a block of statements, or nothing (in the case of empty statements). The **else** clause is optional.

If *expression* evaluates to true (anything other than 0), the statement or block that forms the target of **if** is executed; otherwise, the statement or block that is the target of **else** will be executed, if it exists. Remember, only the code associated with **if** or the code associated with **else** executes, never both.

The conditional statement controlling **if** must produce a scalar result. A *scalar* is an integer, character, pointer, or floating-point type. In C++, it may also be of type **bool**.

PROGRAM: IF

```
#include <iostream>
using namespace std;

int main ()
{
  int magic;                    /* magic number */
  int guess;                    /* user's guess */
  magic = rand ();              /* generate the magic number */
  cout<<"Guess the magic number: "<<endl;
  cin>>guess;
  if (guess == magic)
    cout<<"** Right **";
  return 0;
}
```

OUTPUT:

```
Guess the magic number:
777
```

PROGRAM: IF ELSE IF

```
#include <iostream>
using namespace std;

int main ()
{
  int magic;                    /* magic number */
  int guess;                    /* user's guess */
  magic = rand ();              /* generate the magic number */
  cout << "Guess the magic number: " << endl;
  cin >> guess;
  if (guess == magic)
    cout << "** Right **";
   else
    cout<<"Wrong";
  return 0;
}
```

OUTPUT:

```
Guess the magic number:
897
Wrong
```

PROGRAM: IF –ELSE-IF STATEMENT

```
#include <iostream>
#include<conio.h>
using namespace std;

int main()
{
        int num;
        cout<<"Enter a number: ";
        cin>>num;
        if(num%2==0)
        {
                cout<<"You entered an even number";
        }
        else
        {
                cout<<"You entered an odd number";
        }
        getch();
}
```

OUTPUT:

```
Enter a number: 21
You entered an odd number
```

NESTED ifs

A nested **if** is an **if** that is the target of another **if** or **else**. Nested **if**s are very common in programming. In a nested **if**, an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and that is not already associated with an **else**.

Example,

```
        if(i)
        {
                if(j) statement 1;
                if(k) statement 2; /* this if */
                else statement 3; /* is associated with this else */
        }
        else statement 4; /* associated with if(i) */
```

Standard C++ suggests that at least 256 levels of nested **if**s be allowed in a C++ program which 15 in C language.

PROGRAM: NESTED IF

```
#include <iostream>
using namespace std;

int main ()
{
  int magic;                    /* magic number */
  int guess;                    /* user's guess */
  magic = rand ();              /* generate the magic number */
  cout << "Guess the magic number: " << endl;
  cin >> guess;
  if (guess == magic)
    {
      cout<<"** Right **"<<endl;
      cout<<"  is the magic number\n"<< magic;
    }
  else
    {
      cout<<"Wrong "<<endl;
      if (guess > magic)
           cout<<"too high\n";
      else
           cout<<"too low\n";
    }

    return 0;
}
```

OUTPUT:

```
Guess the magic number:
1721
Wrong
too low
```

The if-else-if Ladder

A common programming construct is the *if-else-if ladder*, sometimes called the *if-else-if staircase* because of its appearance.

Its general form is

```
        if (expression) statement;
        else
        if (expression) statement;
        else
        if (expression) statement;
        ...
        else statement;
```

The conditions are evaluated from the top downward. As soon as a true condition is found, the statement associated with it is executed and the rest of the ladder is bypassed. If none of the conditions are true, the final **else** is executed. That is, if all other conditional tests fail, the last **else** statement is performed. If the final **else** is not present, no action takes place if all other conditions are false.

PROGRAM: IF-ELSE-IF LADDER

```
#include <iostream>
#include<conio.h>
using namespace std;

int main()
{
        char ch;
        float a, b, result;
        cout<<"Enter any two number: ";
        cin>>a>>b;
        cout<<"\n"<<"Enter the operator(+, -, *, /) : ";
        cin>>ch;
        cout<<"\n";
        if(ch=='+')
        {
                result=a+b;
        }
        else if(ch=='-')
        {
                result=a-b;
        }
        else if(ch=='*')
        {
                result=a*b;
        }
        else if(ch=='/')
        {
                result=a/b;
        }
        else
        {
                cout<<"Wrong Operator..!!.. exiting...press a key..";
                getch();
                exit(1);
        }
        cout<<"\n"<<"The calculated result is : "<<result<<"\n";
        getch();
}
```

OUTPUT:

```
Enter any two number: 2
3
Enter the operator(+, -, *, /) : +
The calculated result is : 5
```

PROGRAM 4: IF-ELSE-IF LADDER

```
#include <iostream>
using namespace std;

int main ()
{
  int magic;                    /* magic number */
  int guess;                    /* user's guess */
  magic = rand ();              /* generate the magic number */
  cout << "Guess the magic number: " << endl;
  cin >> guess;
  cout<<magic<<endl;
  if (guess == magic)
    {
      cout<<"** Right ** "<<endl;
      cout<<" is the magic number"<< magic<<endl;
    }
  else if (guess > magic)
    cout<<"Wrong, too high"<<endl;
  else
    cout<<"Wrong, too low"<<endl;
  return 0;
}
```

OUTPUT:

```
Guess the magic number:
67
1804289383
Wrong, too low
```

SWITCH

C++ has a built-in multiple-branch selection statement, called **switch**, which successively tests the value of an expression against a list of integer or character constants. When a match is found, the statements associated with that constant are executed.
The general form of the **switch** statement is

switch (*expression*)

```
{
        case constant1:
                statement sequence
        break;
        case constant2:
                statement sequence
        break;
        case constant3:
                statement sequence
        break;
                ...
        default
                statement sequence
}
```

The *expression* must evaluate to a character or integer value. Floating-point expressions, for example, are not allowed. The value of *expression* is tested, in order, against the values of the constants specified in the **case** statements. When a match is found, the statement sequence associated with that **case** is executed until the **break** statement or the end of the **switch** statement is reached. The **default** statement is executed if no matches are found. The **default** is optional and, if it is not present, no action takes place if all matches fail.

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

There are three important things to know about the switch statement:

- The switch differs from the if in that switch can only test for equality, whereas if can evaluate any type of relational or logical expression.
- No two case constants in the same switch can have identical values. Of course, a switch statement enclosed by an outer switch may have case constants that are the same.
- If character constants are used in the switch statement, they are automatically converted to integers.

The switch statement is often used to process keyboard commands, such as menu selection.

PROGRAM: SWITCH

```
#include <iostream>
#include<conio>
using namespace std;

int main()
{
        int dow;
        cout<<"Enter number of week's day (1-7): ";
        cin>>dow;
        switch(dow)
        {
                case 1 : cout<<"\nSunday";
                        break;
                case 2 : cout<<"\nMonday";
```

```
                    break;
            case 3 : cout<<"\nTuesday";
                    break;
            case 4 : cout<<"\nWednesday";
                    break;
            case 5 : cout<<"\nThursday";
                    break;
            case 6 : cout<<"\nFriday";
                    break;
            case 7 : cout<<"\nSaturday";
                    break;
            default : cout<<"\nWrong number of day";
                    break;
        }
        getch();
}
```

OUTPUT:

```
Enter number of week's day (1-
7): 4

Wednesday
```

NESTED SWITCH STATEMENTS

You can 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 arise. For example, the following code fragment is perfectly acceptable:

```
switch(x)
{
        case 1:
                switch(y)
                {
                        case 0: printf("Divide by zero error.\n");
                        break;
                        case 1: process(x,y);
                }
        break;
        case 2:
        ......
```

.

.COMPARISON OF IF AND SWITCH

The switch differs from the if in that switch can only test for equality, whereas if can evaluate any type of relational or logical expressions.

B) ITERATION STATEMENTS

In C++, and all other modern programming languages, iteration statements (also called *loops*) allow a set of instructions to be executed repeatedly until a certain condition is reached. This condition may be predefined (as in the **for** loop), or open-ended (as in the **while** and **do-while** loops).

1. The for Loop

The general design of the **for** loop is reflected in some form or another in all procedural programming languages. However, in C++, it provides unexpected flexibility and power.

General form:

```
for(initialization; condition; increment)
{
        statement;
}
```

The *initialization* is an assignment statement that is used to set the loop control variable.

The *condition* is a relational expression that determines when the loop exits.

The *increment* defines how the loop control variable changes each time the loop is repeated.

You must separate these three major sections by semicolons.

The **for** loop continues to execute as long as the condition is true. Once the condition becomes false, program execution resumes on the statement following the **for**.

PROGRAM: FOR LOOP

```
#include <iostream>
using namespace std;

int main ()
{
  int x;
  for (x = 1; x <= 10; x++)
    {
      cout << x<<endl;

    }
  return 0;
}
```

OUTPUT:

```
1
2
3
4
5
6
7
8
9
10
```

In **for** loops, the conditional test is always performed at the top of the loop. This means that the code inside the loop may not be executed at all if the condition is false to begin with.

for Loop Variations

Several variations of the **for** are allowed that increase its power, flexibility, and applicability to certain programming situations.

One of the most common variations uses the comma operator to allow two or more variables to control the loop.

For example, the variables **x** and **y** control the following loop, and both are initialized inside the **for** statement:

```
for(x=0, y=0; x+y<10; ++x)
{
        y = getchar();
        y = y - '0'; /* subtract the ASCII code for 0
        from y */
        .
        .
        .
}
```

Commas separate the two initialization statements. Each time the loop repeats, **x** is incremented and **y**'s value is set by keyboard input. Both **x** and **y** must be at the correct value for the loop to terminate. Even though **y**'s value is set by keyboard input, **y** must be initialized to 0 so that its value is defined before the first evaluation of the conditional expression.

PROGRAM: MULTIPLE LOOP VARIABLES

```
#include <iostream>
using namespace std;

int main ()
{
  for (int i = 0, j = 0; i < 3; i++, j++)
    {
      cout << "i: " << i << " j: " << j << endl;
    }
  return 0;
}
```

OUTPUT:

```
i: 0 j: 0
i: 1 j: 1
i: 2 j: 2
```

PROGRAM: PATTERN PRINTING

```
#include <iostream>
#include<conio.h>
using namespace std;

int  main()
{
        int i, j;
        for(i=0; i<5; i++)
        {
                for(j=0; j<=i; j++)
```

```
		{
			cout<<"* ";
		}
		cout<<"\n";
	}
	getch();
}
```

OUTPUT:

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

Interesting trait of the for loop

- The conditional expression does not have to involve testing the loop control variable against some target value. In fact, the condition may be any relational or logical statement. This means that you can test for several possible terminating conditions.
  For example, you could use the following function to log a user onto a remote system. The user has three tries to enter the password. The loop terminates when the three tries are used up or the user enters the correct password.
  ```
  void sign_on(void)
  {
  	char str[20] = "";
  	int x;
  	for(x=0; x<3 && strcmp(str, "password"); ++x)
  	{
  		cout<<"Enter password please:";
  		gets(str);
  	}
  	if(x==3) return;
  		/* else log user in ... */
  }
  ```
  This function uses **strcmp( )**, the standard library function that compares two strings and returns 0 if they match.
- Each of the three sections of the **for** loop may consist of any valid expression. The expressions need not actually have anything to do with what the sections are generally used for.

PROGRAM: FOR LOOP VARIATIONS

```
#include <iostream>
using namespace std;

int sqrnum (int num);
int readnum (void);
int prompt (void);
int
main (void)
{
```

```
  int t;
  for (prompt (); t = readnum (); prompt ())
    {
      sqrnum (t);
    }
  return 0;
}

int
prompt (void)
{
  cout << "Enter a number: " ;
  return 0;
}

int
readnum (void)
{
  int t;
  cin >> t;
  return t;
}

int
sqrnum (int num)
{
  cout << "SQUARE is " <<num * num<<endl;
  return num * num;
}
```

OUTPUT:

```
Enter a number: 17
SQUARE is 289
Enter a number: 21
SQUARE is 441
Enter a number: 0
```

- Pieces of the loop definition need not be there. In fact, there need not be an expression present for any of the sections— the expressions are optional.

  For example, this loop will run until the user enters **123**:

  ```
  for(x=0; x!=123; )
  cout<<x;
  ```

  The increment portion of the **for** definition is blank. This means that each time the loop repeats, **x** is tested to see if it equals 123, but no further action takes place. If you type **123** at the keyboard, however, the loop condition becomes false and the loop terminates.

- The initialization of the loop control variable can occur outside the **for** statement. This most frequently happens when the initial condition of the loop control variable must be computed by some complex means as in this example:

  ```
  gets(s); /* read a string into s */
  ```

```
        if(*s) x = strlen(s); /* get the string's length */
        else x = 10;
        for( ; x<10; )
        {
                cout << x;
                ++x;
        }
```

The initialization section has been left blank and **x** is initialized before the loop is entered.

The Infinite Loop

Although you can use any loop statement to create an infinite loop, **for** is traditionally used for this purpose. Since none of the three expressions that form the **for** loop are required, you can make an endless loop by leaving the conditional expression empty:

```
    for( ; ; )
        cout<<"This loop will run forever.\n";
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C++ programmers more commonly use the **for(;;)** construct to signify an infinite loop.

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. Program control then resumes at the code following the loop, as shown here:

```
    ch = '\0';
    for( ; ; )
    {
        ch = getchar(); /* get a character */
        if(ch=='A') break; /* exit the loop */
    }
    cout<<"you typed an A";
```

This loop will run until the user types an **A** at the keyboard.

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.

Removing spaces from an input stream is a common programming task. For example, a database program may allow a query such as "show all balances less than 400." The database needs to have each word fed to it separately, without leading spaces. That is, the database input processor recognizes "**show**" but not " **show**". The following loop shows one way to accomplish this. It advances past leading spaces in the string pointed to by **str**.

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

As you can see, this loop has no body—and no need for one either.

Time delay loops are often used in programs. The following code shows how to create one by using **for**:

```
     for(t=0; t<SOME_VALUE; t++) ;
```

2. **The while Loop**

The second loop available in C/C++ is the **while** loop.

General form:

```
    while(condition)
```

```
{
        statement;
}
```

where *statement* is either an empty statement, a single statement, or a block of statements.

The *condition* may be any expression, and true is any nonzero value. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line of code immediately following the loop.

The following example shows a keyboard input routine that simply loops until the user types **A**:

```
char wait_for_char(void)
{
        char ch;
        ch = '\0'; /* initialize ch */
        while(ch != 'A') ch = getchar();
        return ch;
}
```

while loops check the test condition at the top of the loop, which means that the body of the loop will not execute if the condition is false to begin with. This feature may eliminate the need to perform a separate conditional test before the loop.

PROGRAM : WHILE LOOP

```
#include <iostream>
#include<conio.h>
using namespace std;

int main()
{
        unsigned long num, fact=1;
        cout<<"Enter a number: ";
        cin>>num;
        while(num)
        {
                fact = fact*num;
                num--;
        }
        cout<<"The factorial of the number is "<<fact;
        getch();
}
```

OUTPUT:

```
Enter a number: 34
The factorial of the number is 4926277576697053184
```

PROGRAM 20: CHECK PALINDROME OR NOT

```
#include <iostream>
#include<conio.h>
```

```
using namespace std;

int  main()
{
        int num, rem, orig, rev=0;
        cout<<"Enter a number : ";
        cin>>num;
        orig=num;
        while(num!=0)
        {
                rem=num%10;
                rev=rev*10 + rem;
                num=num/10;
        }
        if(rev==orig)  // check if original number is equal to its reverse
        {
                cout<<"Palindrome";
        }
        else
        {
                cout<<"Not Palindrome";
        }
        getch();
}
```

OUTPUT:

```
Enter a number : 2345
Not Palindrome
```

Interesting trait of the while loop

- If several separate conditions need to terminate a **while** loop, a single variable commonly forms the conditional expression. The value of this variable is set at various points throughout the loop.
  In this example,

  ```
  void func1(void)
  {
          int working;
          working = 1; /* i.e., true */
          while(working)
          {
          working = process1();
          if(working)
          working = process2();
          if(working)
          working = process3();
          }
  }
  ```

  Any of the three routines may return false and cause the loop to exit.
- There need not be any statements in the body of the **while** loop.
  For example,

```
while((ch=getchar()) != 'A') ;
```

will simply loop until the user types **A**. If you feel uncomfortable putting the assignment inside the **while** conditional expression, remember that the equal sign is just an operator that evaluates to the value of the right-hand operand.

3. The do-while Loop

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do-while** loop checks its condition at the bottom of the loop. This means that a **do-while** loop always executes at least once.

General form:

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

The **do-while** loop iterates until *condition* becomes false.

Perhaps the most common use of the **do-while** loop is in a menu selection function. When the user enters a valid response, it is returned as the value of the function. Invalid responses cause a reprompt.

PROGRAM: DO-WHILE

```
#include <iostream>
#include<conio.h>
using namespace std;

int main()
{

        int num, l=0;
        cout<<"Enter a number: ";
        cin>>num;
        cout<<"\nIncrementing & Printing the number, 10 times:\n";
        do
        {
                num++;
                cout<<num<<"\n";
                l++;
        }while(l<10);
        getch();
}
```

OUTPUT:

```
Enter a number: 04
Incrementing & Printing the number, 10 times:

5
6
7
8
```

```
9
10
11
12
13
14
```

PROGRAM: FINDING AREA, PERIMETER AND DIAGONAL OF A RECTANGLE

```
#include <iostream>
#include<conio.h>
#include<math.h>
using namespace std;

int main()
{
	char ch, ch1;
	float l, b, peri, area, diag;
	cout<<"Rectangle Menu";
	cout<<"\n 1. Area";
	cout<<"\n 2. Perimeter";
	cout<<"\n 3. Diagonal";
	cout<<"\n 4. Exit\n";
	cout<<"\nEnter your choice: ";
	do
	{
		cin>>ch;
		if(ch == '1' || ch == '2' || ch == '3')
		{
			cout<<"Enter length & breadth: ";
			cin>>l>>b;
		}
		switch(ch)
		{
			case '1' : area = l * b ;
				cout<<"Area = "<<area;
				break ;
			case '2' : peri = 2 * (l + b);
				cout<<"Perimeter = "<<peri;
				break;
			case '3' : diag = sqrt((l * l) + (b * b));
				cout<<"Diagonal = "<<diag;
				break;
			case '4' : cout<<"Breaking..Press a key..";
				getch();
				exit(1);
			default : cout<<"Wrong choice !!!!";
				cout<<"\nEnter a valid one";
				break;
		}	//end of switch
		cout<<"\nWant to enter more (y/n) ? ";
```

```
                cin>>ch1;
                if(ch1 == 'y' || ch1 == 'Y')
                cout<<"Again enter choice (1-4): ";
        }while(ch1 == 'y' || ch1 == 'Y') ;      //end of DO-WHILE loop
        getch();
}
```

OUTPUT:

```
Rectangle Menu
 1. Area
 2. Perimeter
 3. Diagonal
 4. Exit

Enter your choice: 1
Enter length & breadth: 2
3
Area = 6
Want to enter more (y/n) ? n
```

DECLARING VARIABLES WITHIN SELECTION AND ITERATION STATEMENTS

In C++ (but not C89), it is possible to declare a variable within the conditional expression of an **if** or **switch**, within the conditional expression of a **while** loop, or within the initialization portion of a **for** loop. A variable declared in one of these places has its scope limited to the block of code controlled by that statement. For example, a variable declared within a **for** loop will be local to that loop.

Example: declares a variable within the initialization portion of a **for** loop:

```
        /* i is local to for loop; j is known outside loop. */
int j;
for(int i = 0; i<10; i++)
j = i * i;
/* i = 10; // *** Error *** -- i not known here! */
```

Here, **i** is declared within the initialization portion of the **for** and is used to control the loop. Outside the loop, **i** is unknown.

C++, then you can also declare a variable within any conditional expression, such as those used by the **if** or a **while**.

Example: ,

```
if(int x = 20)
{
        x = x - y;
        if(x>10) y = 0;
}
```

declares **x** and assigns it the value 20. Since this is a true value, the target of the **if** executes. Variables declared within a conditional statement have their scope limited to the block of code controlled by that statement. Thus, in this case, **x** is not known outside the **if**.

C) JUMP STATEMENTS

C++ has four statements that perform an unconditional branch: **return**, **goto**, **break**, and **continue**. Of these, you may use **return** and **goto** anywhere in your program. You may use the **break** and **continue** statements in conjunction with any of the loop statements.

1. **The return Statement**

The **return** statement is used to return from a function. It is categorized as a jump statement because it causes execution to return (jump back) to the point at which the call to the function was made. A **return** may or may not have a value associated with it. If **return** has a value associated with it, that value becomes the return value of the function.

In C89, a non-**void** function does not technically have to return a value. If no return value is specified, a garbage value is returned. However, in C++ (and in C99), a non-**void** function *must* return a value. That is, in C++, if a function is specified as returning a value, any **return** statement within it must have a value associated with it.

General form: return *expression*;

The *expression* is present only if the function is declared as returning a value. In this case, the value of *expression* will become the return value of the function.

You can use as many **return** statements as you like within a function. However, the function will stop executing as soon as it encounters the first **return**. The **}** that ends a function also causes the function to return. It is the same as a **return** without any specified value. If this occurs within a non-**void** function, then the return value of the function is undefined.

A function declared as **void** may not contain a **return** statement that specifies a value. Since a **void** function has no return value, it makes sense that no **return** statement within a **void** function can return a value.

2. **The goto Statement**

It is used for jumping to a specific location. The **goto** statement requires a label for operation. (A *label* is a valid identifier followed by a colon.) Furthermore, the label must be in the same function as the **goto** that uses it—you cannot jump between functions.

General form:

statement is
goto *label*;
...
label:
where *label* is any valid label either before or after **goto**.

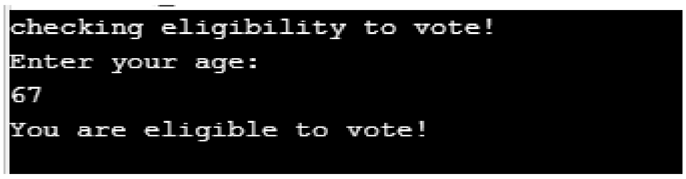
Example: you could create a loop from 1 to 100 using the **goto** and a label, as shown here:

```
x = 1;
loop1:
x++;
if(x<100) goto loop1;
```

PROGRAM: GOTO

```
#include <iostream>
using namespace std;

int main()
{
  ineligible:
    cout<<"checking eligibility to vote!\n";
    cout<<"Enter your age:\n";
    int age;
    cin>>age;
    if (age < 18){
            goto ineligible;
    }
    else
    {
            cout<<"You are eligible to vote!";
    }
}
```

OUTPUT: _



```
checking eligibility to vote!
Enter your age:
67
You are eligible to vote!
```

3. **The break Statement**

The **break** statement has two uses.

- You can use it to terminate a **case** in the **switch** statement.
- You can also use it to force immediate termination of a loop, bypassing the normal loop conditional test.

When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.

For example,

```
#include <iostream>
using namespace std;

int main(void)
{
int t;
for(t=0; t<100; t++) {
cout<< t;
if(t==10) break;
}
return 0;
}
```

Programmers often use the **break** statement in loops in which a special condition can cause immediate termination.

A **break** causes an exit from only the innermost loop.

Example:

```
        for(t=0; t<100; ++t)
        {
                count = 1;
                for(;;)
                {
                        cout<<count;
                        count++;
                        if(count==10) break;
                }
        }
```

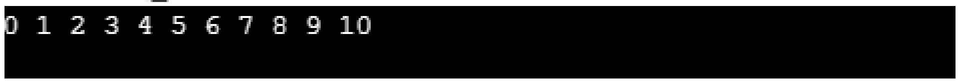
prints the numbers 1 through 10 on the screen 100 times. Each time execution encounters break, control is passed back to the outer **for** loop.

A **break** used in a **switch** statement will affect only that **switch**. It does not affect any loop the **switch** happens to be in.

PROGRAM: BREAK

```
#include <iostream>
using namespace std;

int main(void)
{
int t;
for(t=0; t<100; t++)
{
printf("%d ", t);
if(t==10) break;
}
return 0;

}
```

OUTPUT:



```
0 1 2 3 4 5 6 7 8 9 10
```

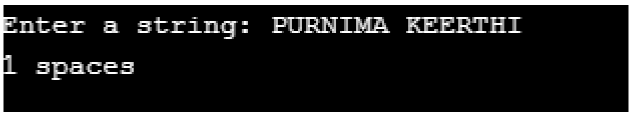
4. The continue Statement

The **continue** statement works somewhat like the **break** statement. Instead of forcing termination, however, **continue** forces the next iteration of the loop to take place, skipping any code in between. For the **for** loop, **continue** causes the conditional test and increment portions of the loop to execute. For the **while** and **do-while** loops, program control passes to the conditional tests. For example, the following program counts the number of spaces contained in the string entered by the user:

PROGRAM: CONTINUE

```
include <iostream>
using namespace std;

int main(void)
{
char s[80], *str;
int space;
printf("Enter a string: ");
gets(s);
str = s;
for(space=0; *str; str++)
{
if(*str != ' ') continue;
space++;
}
printf("%d spaces\n", space);
return 0;
}
```

OUTPUT:



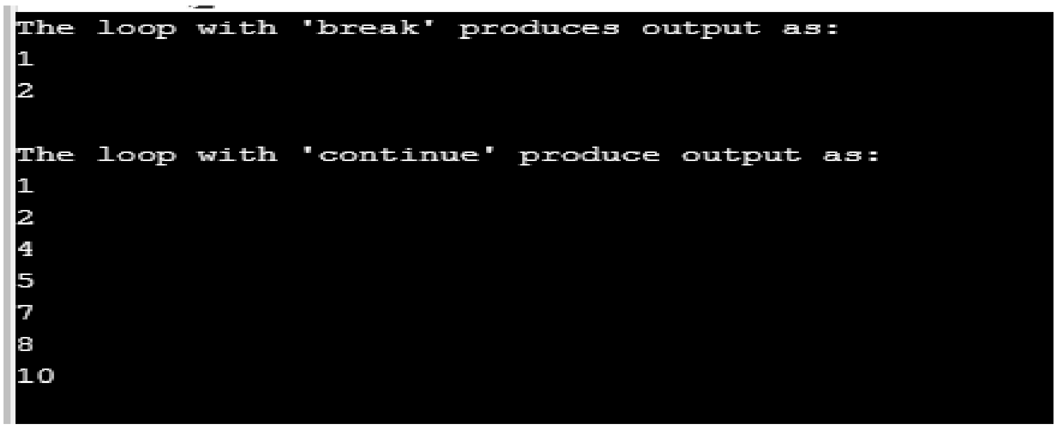
```
Enter a string: PURNIMA KEERTHI
1 spaces
```

Each character is tested to see if it is a space. If it is not, the **continue** statement forces the **for** to iterate again. If the character *is* a space, **space** is incremented.

PROGRAM: BREAK AND CONTINUE

```
#include <iostream>
using namespace std;

int main()
{
  cout<<"The loop with \'break\' produces output as:\n";
        for(int i=1; i<=10; i++)
        {
                if((i%3)==0)
                        break;
                else
                        cout<<i<<endl;
        }
        cout<<"\nThe loop with \'continue\' produce output as:\n";
        for(int i=1; i<=10; i++)
        {
                if((i%3)==0)
                        continue;
                else
                        cout<<i<<endl;
        }

}
```

OUTPUT:



```
The loop with 'break' produces output as:
1
2

The loop with 'continue' produce output as:
1
2
4
5
7
8
10
```

ARRAYS

Definition: 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++, all arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Single-Dimension Arrays

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

General form: *type var_name[size]*;

Like other variables, arrays must be explicitly declared so that the compiler may allocate space for them in memory. Here, *type* declares the base type of the array, which is the type of each element in the array, and *size* defines how many elements the array will hold.

Example: To declare a 100-element array called balance of type double, use this statement:

```
double balance[100];
```

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array.

Example: balance[3] = 12.23;

assigns element number 3 in balance the value 12.23.

In C++, all arrays have 0 as the index of their first element. Therefore, when you write

```
char p[10];
```

you are declaring a character array that has ten elements, p[0] through p[9].

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:

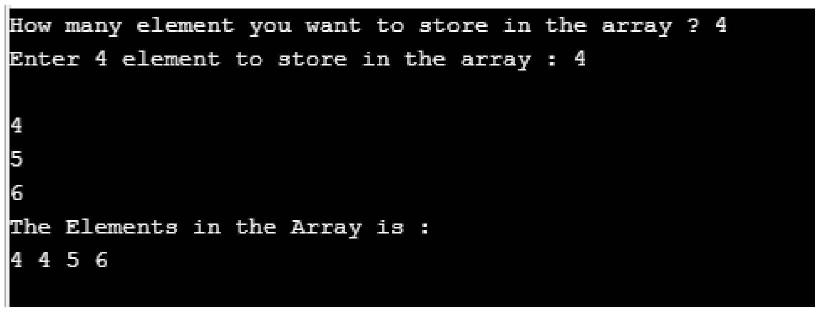
total bytes = sizeof(base type) x size of array

C++ 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.

PROGRAM 31: SINGLE DIMENSIONAL ARRAY

```
#include <iostream>
using namespace std;

int main()
{
  int arr[50], n;
        cout<<"How many element you want to store in the array ? ";
        cin>>n;
        cout<<"Enter "<<n<<" element to store in the array : ";
        for(int i=0; i<n; i++)
        {
                cin>>arr[i];
        }
        cout<<"The Elements in the Array is : \n";
        for(int  i=0; i<n; i++)
        {
                cout<<arr[i]<<" ";
        }
}
```

OUTPUT:

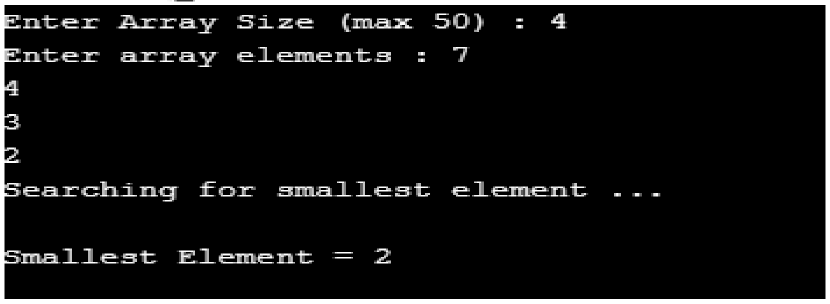


```
How many element you want to store in the array ? 4
Enter 4 element to store in the array : 4

4
5
6
The Elements in the Array is :
4 4 5 6
```

PROGRAM 32: LARGEST ELEMENTS IN ARRAY

```
#include <iostream>
using namespace std;

int main()
{
  int small, arr[50], size, i;
        cout<<"Enter Array Size (max 50) : ";
        cin>>size;
        cout<<"Enter array elements : ";
        for(i=0; i<size; i++)
        {
                cin>>arr[i];
        }
        cout<<"Searching for smallest element ...\n\n";
        small=arr[0];
        for(i=0; i<size; i++)
        {
                if(small>arr[i])
                {
                        small=arr[i];
                }
        }
        cout<<"Smallest Element = "<<small;
}
```

OUTPUT:



```
Enter Array Size (max 50) : 4
Enter array elements : 7
4
3
2
Searching for smallest element ...

Smallest Element = 2
```

Two-Dimensional Arrays

C++ supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, essentially, an array of one-dimensional arrays.

To declare a two-dimensional integer array d of size 10,20, you would write

int d[10][20];

C++ places each dimension in its own set of brackets.

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)

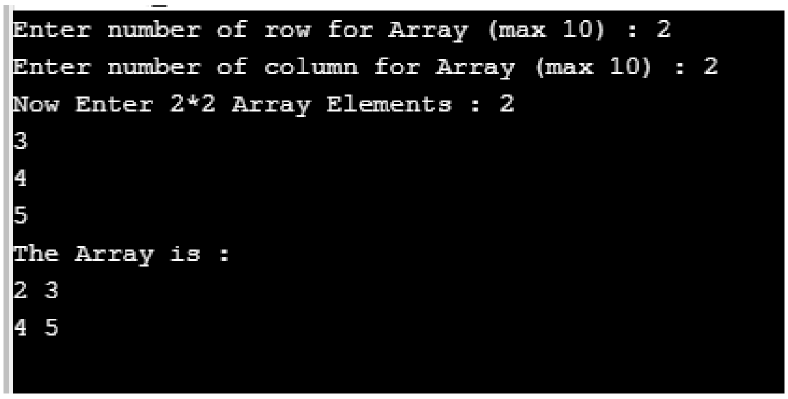
Therefore, assuming 4-byte integers, an integer array with dimensions 10,5 would have 10 x 5 x 4 or 200 bytes allocated.

Two-dimensional arrays are stored in a row-column matrix, where the first index indicates the row and the second indicates the column. This means that the rightmost index changes faster than the leftmost when accessing the elements in the array in the order in which they are actually stored in memory.

PROGRAM 38: TWO DIMENSIONAL ARRAYS

```
#include <iostream>
using namespace std;

int main()
{
  int arr[10][10], row, col, i, j;
        cout<<"Enter number of row for Array (max 10) : ";
        cin>>row;
        cout<<"Enter number of column for Array (max 10) : ";
        cin>>col;
        cout<<"Now Enter "<<row<<"*"<<col<<" Array Elements : ";
        for(i=0; i<row; i++)
        {
                for(j=0; j<col; j++)
                {
                        cin>>arr[i][j];
                }
        }
        cout<<"The Array is :\n";
        for(i=0; i<row; i++)
        {
                for(j=0; j<col; j++)
                {
                        cout<<arr[i][j]<<" ";
                }
                cout<<"\n";
        }
}
```

OUTPUT: _



```
Enter number of row for Array (max 10) : 2
Enter number of column for Array (max 10) : 2
Now Enter 2*2 Array Elements : 2
3
4
5
The Array is :
2 3
4 5
```

PROGRAM 39: ADD TWO MATRICES

```
#include <iostream>
using namespace std;

int main()
{
   int r, c, a[100][100], b[100][100], sum[100][100], i, j;
    cout << "Enter number of rows (between 1 and 100): ";
    cin >> r;
    cout << "Enter number of columns (between 1 and 100): ";
    cin >> c;
    cout << endl << "Enter elements of 1st matrix: " << endl;
    // Storing elements of first matrix entered by user.
    for(i = 0; i < r; ++i)
       for(j = 0; j < c; ++j)
       {
           cout << "Enter element a" << i + 1 << j + 1 << " : ";
           cin >> a[i][j];
       }
    // Storing elements of second matrix entered by user.
    cout << endl << "Enter elements of 2nd matrix: " << endl;
    for(i = 0; i < r; ++i)
       for(j = 0; j < c; ++j)
       {
           cout << "Enter element b" << i + 1 << j + 1 << " : ";
           cin >> b[i][j];
       }
    // Adding Two matrices
    for(i = 0; i < r; ++i)
        for(j = 0; j < c; ++j)
            sum[i][j] = a[i][j] + b[i][j];
    // Displaying the resultant sum matrix.
    cout << endl << "Sum of two matrix is: " << endl;
    for(i = 0; i < r; ++i)
        for(j = 0; j < c; ++j)
        {
            cout << sum[i][j] << "  ";
            if(j == c - 1)
            cout << endl;
      }
   return 0;
}
```

OUTPUT:

```
Enter number of rows (between 1 and 100): 2
Enter number of columns (between 1 and 100): 2

Enter elements of 1st matrix:
Enter element a11 : 1
Enter element a12 : 2
Enter element a21 : 3
Enter element a22 : 4

Enter elements of 2nd matrix:
Enter element b11 : 4
5
Enter element b12 : Enter element b21 : 7
Enter element b22 : 7

Sum of two matrix is:
5  7
10  11
```

Multidimensional Arrays

C++ allows arrays of more than two dimensions. The exact limit, if any, is determined by your compiler.

General form: *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. If the array held 2-byte integers, 4,320 bytes would be needed. If the array held doubles (assuming 8 bytes per double), 17,280 bytes would be required.

The storage required increases exponentially with the number of dimensions. For example, if a fifth dimension of size 10 was added to the preceding array, then 172,
800 bytes would be required.

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.

Array Initialization

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

General form: *type_specifier array_name*[*size1*]. . .[*sizeN*] = { *value_list* };

The *value_list* is a comma-separated list of values whose type is compatible with *type_specifier*. The first value is placed in the first position of the array, the second value in the second position, and so on. Note that a semicolon follows the }.

Example: A 10-element integer array is initialized with the numbers 1 through 10:

```
int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```
This means that i[0] will have the value 1 and i[9] will have the value 10.

Character arrays that hold strings allow a shorthand initialization that takes the form:

char *array_name*[*size*] = "*string*";

Example, this code fragment initializes str to the phrase "I like C++".
```
char str[11] = "I like C++";
```
This is the same as writing char str[11] = `{'I', ' ', 'l', 'i', 'k', 'e',' ', 'C', '+', '+', '\0'};`

Multidimensional arrays are initialized the same as single-dimension ones.

Example: The following initializes sqrs with the numbers 1 through 10 and their squares.
```
int sqrs[10][2] = {1, 1,
                   2, 4,
                   3, 9,
                   4, 16,
                   5, 25,
                   6, 36,
                   7, 49,
                   8, 64,
                   9, 81,
                  10, 100
                  };
```
When initializing a multidimensional array, you may add braces around the initializers for each dimension. This is called *subaggregate grouping*.

Example: Here is another way to write the preceding declaration.
```
int sqrs[10][2] = { {1, 1},
                    {2, 4},
                    {3, 9},
                    {4, 16},
                    {5, 25},
                    {6, 36},
                    {7, 49},
                    {8, 64},
                    {9, 81},
                    {10, 100}
                  };
```
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

Imagine that you are using array initialization to build a table of error messages, as shown here:
```
char e1[12] = "Read error\n";
char e2[13] = "Write error\n";
char e3[18] = "Cannot open file\n";
```
As you might guess, it is tedious to count the characters in each message manually\ to determine the correct array dimension. Fortunately, you can let the compiler automatically calculate the dimensions of the arrays. If, in an array initialization statement, the size of the array is not specified, the C++ compiler automatically creates an array big enough to hold all

the initializers present. This is called an *unsized array*. Using this approach, the message table becomes

```
char e1[] = "Read error\n";
char e2[] = "Write error\n";
char e3[] = "Cannot open file\n";
```

Given these initializations, this statement

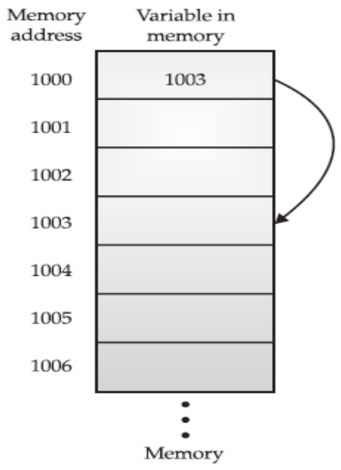
```
cout<<"has length \n", e2, sizeof(e2);
```

will print

```
Write error has length 13
```

POINTERS

Definition: A *pointer* is a variable that holds a memory address. This address is the location of another object (typically another variable) in memory. For example, if one variable contains the address of another variable, the first variable is said to *point to* the second.



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.

General form: *type *name;*

where *type* is the base type of the pointer and may be any valid type. The name of the pointer variable is specified by *name*.

The Pointer Operators

There are two special pointer operators: * and &.

& Operator

The & is a unary operator that returns the memory address of its operand.

Example: m = &count;

places into m the memory address of the variable count. This address is the computer's internal location of the variable. It has nothing to do with the value of count. You can think of & as returning "the address of." Therefore, the preceding assignment statement means "m receives the address of count."

To understand the above assignment better, assume that the variable count uses memory location 2000 to store its value. Also assume that count has a value of 100. Then, after the preceding assignment, m will have the value 2000.

*** Operator**

The second pointer operator, *, is the complement of &. It is a unary operator that returns the value located at the address that follows.

Example: If m contains the memory address of the variable count,

```
q = *m;
```

places the value of count into q. Thus, q will have the value 100 because 100 is stored at location 2000, which is the memory address that was stored in m. You can think of * as "at address." In this case, the preceding statement means "q receives the value at address m."

In C++, it is illegal to convert one type of pointer into another without the use of an explicit type cast.

PROGRAM 43: POINTERS

```
#include <iostream>
using namespace std;

int main()
{
    int var1 = 3;
   int var2 = 24;
   int var3 = 17;
   cout << &var1 << endl;
   cout << &var2 << endl;
   cout << &var3 << endl;
}
```

OUTPUT:

```
0x7ffe74f8b134
0x7ffe74f8b138
0x7ffe74f8b13c
```

PROGRAM 44: POINTERS

```
#include <iostream>
using namespace std;

int main()
{
   int *pc, c;

   c = 5;
   cout << "Address of c (&c): " << &c << endl;
   cout << "Value of c (c): " << c << endl << endl;
   pc = &c;    // Pointer pc holds the memory address of variable c
   cout << "Address that pointer pc holds (pc): "<< pc << endl;
   cout << "Content of the address pointer pc holds (*pc): " << *pc << endl << endl;

   c = 11;    // The content inside memory address &c is changed from 5 to 11.
   cout << "Address pointer pc holds (pc): " << pc << endl;
   cout << "Content of the address pointer pc holds (*pc): " << *pc << endl << endl;
   *pc = 2;
```

```
    cout << "Address of c (&c): " << &c << endl;
    cout << "Value of c (c): " << c << endl << endl;
    return 0;
}
```

OUTPUT:

```
Address of c (&c): 0x7ffe72819c04
Value of c (c): 5

Address that pointer pc holds (pc): 0x7ffe72819c04
Content of the address pointer pc holds (*pc): 5

Address pointer pc holds (pc): 0x7ffe72819c04
Content of the address pointer pc holds (*pc): 11

Address of c (&c): 0x7ffe72819c04
Value of c (c): 2
```

Pointer Expressions

1. **Pointer Assignments**

As with any variable, you may use a pointer on the right-hand side of an assignment statement to assign its value to another pointer.

PROGRAM 47: POINTER ASSIGNMENTS

```
#include <iostream>
using namespace std;

int main()
{
  int x;
int *p1, *p2;
p1 = &x;
p2 = p1;
cout<<p2; /* print the address of x, not x's value! */
return 0;
}
```

OUTPUT:

```
0x7ffd862f1e5c
```

2. **Pointer Arithmetic**

There are only two arithmetic operations that you may use on pointers: addition and subtraction. To understand what occurs in pointer arithmetic, let p1 be an integer pointer with a current value of 2000. Also, assume integers are 2 bytes long.

After the expression

p1++;

p1 contains 2002, not 2001. The reason for this is that each time p1 is incremented, it will point to the next integer. The same is true of decrements. For example, assuming that p1 has the value 2000, the expression

```
p1--;
```

causes p1 to have the value 1998.

You are not limited to the increment and decrement operators. For example, you may add or subtract integers to or from pointers. The expression

```
p1 = p1 + 12;
```

makes p1 point to the twelfth element of p1's type beyond the one it currently points to.

Besides addition and subtraction of a pointer and an integer, only one other arithmetic operation is allowed: You may subtract one pointer from another in order to find the number of objects of their base type that separate the two. All other arithmetic operations are prohibited. Specifically, you may not multiply or divide pointers; you may not add two pointers; you may not apply the bitwise operators to them; and you may not add or subtract type float or double to or from pointers.

PROGRAM 48: POINTER ARITHMETIC: INCREMENTING POINTERS

```
#include <iostream>
using namespace std;

int main()
{
   int var[5] = {10,20,30,40,50}; //Array Declaration.
        int *ptr; //pointer point to int.

        ptr = var; // let us have array address in pointer.

        for (int i = 0; i < 5; i++) //Loop to show Address and Value:
        {
          cout << "\n Address of var[" << i << "] = " <<ptr<<endl; //show the Address:
          cout << "Value of var[" << i << "] = "<<*ptr<<endl; //show the value:

       ptr++; // point to the next location (incrementation)
        }


    return 0;
  }
```

OUTPUT:

```
Address of var[0] = 0x7fff8ea2bc90
Value of var[0] = 10

 Address of var[1] = 0x7fff8ea2bc94
Value of var[1] = 20

 Address of var[2] = 0x7fff8ea2bc98
Value of var[2] = 30
```

```
 Address of var[3] = 0x7fff8ea2bc9c
Value of var[3] = 40

 Address of var[4] = 0x7fff8ea2bca0
Value of var[4] = 50
```

PROGRAM 49: POINTER ARITHMETIC: DECREMENTING POINTERS

```
#include <iostream>
using namespace std;

int main()
{
   int var[5] = {10,20,30,40,50}; //Array Declaration.
   int *ptr; //pointer point to int.

   ptr = &var[4]; //ptr point to the last address of Array:

   for (int i = 5; i > 0; i--) //Loop to show Address and Value:
   {
         cout << "\n Address of var[" << i << "] = " <<ptr<<endl; //show the Address:
         cout << "Value of var[" << i << "] = "<<*ptr<<endl; //show the value:

         ptr--; // point to the Previous location:
    }
   return 0;
  }
```

OUTPUT:

```
Address of var[5] = 0x7ffec8e97200
Value of var[5] = 50

 Address of var[4] = 0x7ffec8e971fc
Value of var[4] = 40

 Address of var[3] = 0x7ffec8e971f8
Value of var[3] = 30

 Address of var[2] = 0x7ffec8e971f4
Value of var[2] = 20

 Address of var[1] = 0x7ffec8e971f0
Value of var[1] = 10
```

3. 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)
cout<<"p points to lower memory than q\n";
```

Generally, pointer comparisons are used when two or more pointers point to a common object, such as an array

PROGRAM 50: POINTER COMPARISION

```
#include <iostream>
using namespace std;

int main()
{
    int var[5] = {10,20,30,40,50}; //Array Declaration.
   int *ptr; //pointer point to int.

   ptr = var; //ptr point to the first address of Array:
   int i=0; //Variable to use in while loop:

   while(ptr <= &var[4]) //Loop to show Address and Value:
   {
      for(int i=0; i<5; i++)
      {
            cout << "\n Address of var[" << i << "] = "
                 <<ptr<<endl; //show the Address:

            cout << "Value of var[" << i << "] = "
                 <<*ptr<<endl; //show the value:

            ptr++; // point to the Next location:
       }
    }
     return 0;
   }
```

OUTPUT:

```
Address of var[0] = 0x7ffe055b21f0
Value of var[0] = 10

 Address of var[1] = 0x7ffe055b21f4
Value of var[1] = 20

 Address of var[2] = 0x7ffe055b21f8
Value of var[2] = 30

 Address of var[3] = 0x7ffe055b21fc
Value of var[3] = 40

 Address of var[4] = 0x7ffe055b2200
Value of var[4] = 50
```

Pointers and Arrays

There is a close relationship between pointers and arrays. Consider this program fragment:

```
char str[80], *p1;
p1 = str;
```

Here, p1 has been set to the address of the first array element in str. To access the fifth element in str, you could write

```
str[4]
or
*(p1+4)
```

C++ provides two methods of accessing array elements: pointer arithmetic and array indexing. Although the standard array-indexing notation is sometimes easier to understand, pointer arithmetic can be faster. Since speed is often a consideration in programming, C/C++ programmers commonly use pointers to access array elements.

These two versions of putstr( )—one with array indexing and one with pointers

```
/* Index s as an array. */
void putstr(char *s)
{
register int t;
for(t=0; s[t]; ++t) putchar(s[t]);
}

/* Access s as a pointer. */
void putstr(char *s)
{
while(*s) putchar(*s++);
}
```

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]
```

Initializing Pointers

After a nonstatic local pointer is declared but before it has been assigned a value, it contains an unknown value. There is an important convention that most C/C++ programmers follow when working with pointers: A pointer that does not currently point to a valid memory location is given the value null (which is zero).

All pointers, when they are created, should be initialized to some value, even if it is only zero. A pointer whose value is zero is called a null pointer.

If the pointer is initialized to zero, you must specifically assign the address to the pointer.

```
pCount = &Count; // assign the address to the pointer (NO * is present)
```

It is also possible to assign the address at the time of declaration.

```
int *pCount = &Count;  //declare and assign an integer pointer
```

Another variation on the initialization theme is the following type of string declaration:

```
char *p = "hello world";
```

PROGRAM 51: ARRAY OF POINTERS: LAB PROGRAM

FUNCTIONS

Functions are the building blocks of C and C++ and the place where all program activity occurs

General Form:

ret-type function-name(*parameter list*)
{
body of the function
}

The *ret-type* specifies the type of data that the function returns. A function may return any type of data except an array. The *parameter list* is a comma-separated list of variable names and their associated types that receive the values of the arguments when the function is called. A function may be without parameters

All function parameters must be declared individually, each including both the type and name.

General form:

f(*type varname1, type varname2, . . . , type varnameN)*

For example,

```
f(int i, int k, int j) /* correct */
f(int i, k, float j) /* incorrect */
```

PROGRAM 52: FUNCTIONS

```
#include <iostream>
using namespace std;

int add(int, int);
int main()
{
   int num1, num2, sum;
   cout<<"Enters two numbers to add: ";
   cin >> num1 >> num2;
   // Function call
   sum = add(num1, num2);
   cout << "Sum = " << sum;
   return 0;
}
// Function definition
int add(int a, int b)
{
   int add;
   add = a + b;
```

```
    // Return statement
    return add;
}
```

OUTPUT:

```
Enters two numbers to add: 5
4
Sum = 9
```

Scope Rules of Functions

The *scope rules* of a language are the rules that govern whether a piece of code knows about or has access to another piece of code or data. Each function is a discrete block of code. A function's code is private to that function and cannot be accessed by any statement in any other function except through a call to that function because the two functions have a different scope.

Variables that are defined within a function are called *local* variables. A local variable comes into existence when the function is entered and is destroyed upon exit. That is, local variables cannot hold their value between function calls.

In C++ you cannot define a function within a function.

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function. They behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

```
        /* Return 1 if c is part of string s; 0 otherwise. */
int is_in(char *s, char c)
{
        while(*s)
        if(*s==c)   return 1;
        else   s++;
        return 0;
 }
```

Parameter Passing

In a computer language, there are two ways that arguments can be passed to a subroutine.

a. ***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.

By default, C/C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function.

PROGRAM 53: CALL BY VALUE

```
#include <iostream>
#include<conio.h>
using namespace std;

void swap(int a, int b)
```

```
{
   int temp;
   temp = a;
   a = b;
   b = temp;
}

int main()
{
   int a = 100, b = 200;

   swap(a, b);  // passing value to function
   cout<<"Value of a"<<a;
   cout<<"Value of b"<<b;
   getch();
   return 0;
}
```

OUTPUT:

```
Value of a100Value of b200
```

b. Call by reference

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.

Creating a Call by Reference

You can create a call by reference by passing a pointer to an argument, instead of the argument itself. Since the address of the argument is passed to the function, code within the function can change the value of the argument outside the function. Pointers are passed to functions just like any other value.

Example:

```
        void swap(int *x, int *y)
        {
                int temp;
                temp = *x; /* save the value at address x */
                *x = *y; /* put y into x */
                *y = temp; /* put x into y */
        }
```

swap( ) is able to exchange the values of the two variables pointed to by **x** and **y** because their addresses (not their values) are passed. Thus, within the function, the contents of the variables can be accessed using standard pointer operations, and the contents of the variables used to call the function are swapped.

PROGRAM 54: CALL BY REFERENCE

```
#include<iostream>
using namespace std;
#include<conio.h>

void swap (int *a, int *b)
```

```
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}

int main ()
{
  int a = 100, b = 200;

  swap (&a, &b);                  // passing value to function
  cout << "Value of a" << a;
  cout << "Value of b" << b;
  return 0;
}
```

OUTPUT:

```
Value of a200Value of b100
```

C++ allows you to fully automate a call by reference through the use of reference parameters.

Reference parameters: refer References

Function declaration (Function Prototypes)

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

General form

type func_name(type parm_name1, type parm_name2,. . .,
type parm_nameN);

Example: void sqr_it(int *i); /* prototype */

The only function that does not require a prototype is **main( ),** since it is the first function called when your program begins.

In C++, an empty parameter list is simply indicated in the prototype by the absence of any parameters.

Example: int f(); /* C++ prototype for a function with no parameters */

However, in C this prototype means something different. An empty parameter list simply says that *no parameter information* is given.

In C, when a function has no parameters, its prototype uses **void** inside the parameter list.

Example: here is **f( )**'s prototype as it would appear in a C program.

float f(void);

This tells the compiler that the function has no parameters, and any call to that function that has parameters is an error. In C++, the use of **void** inside an empty parameter list is still allowed, but is redundant.

*In C++, **f( )** and **f(void)** are equivalent.*

Default Function Arguments

C++ allows a function to assign a parameter a default value when no argument corresponding to that parameter is specified in a call to that function. The default value is specified in a manner syntactically similar to a variable initialization.

Example: This declares **myfunc( )** as taking one **double** argument with a default value of 0.0:

```
void myfunc(double d = 0.0)
{
        // ...
}
```

Now, **myfunc( )** can be called one of two ways, as the following examples show:

```
myfunc(198.234); // pass an explicit value
myfunc(); // let function use default
```

The first call passes the value 198.234 to **d**. The second call automatically gives **d** the default value zero.

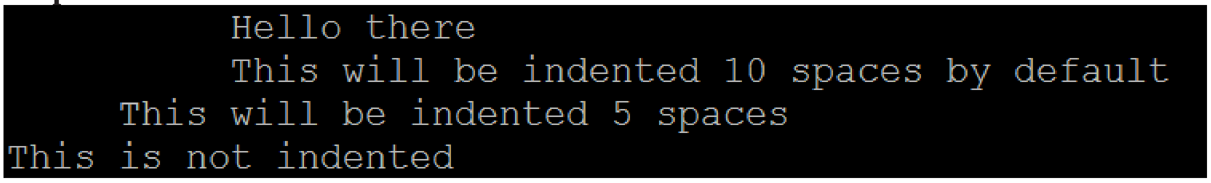
One reason that default arguments are included in C++ is because they provide another method for the programmer to manage greater complexity. To handle the widest variety of situations, quite frequently a function contains more parameters than are required for its most common usage. Thus, when the default arguments apply, you need specify only the arguments that are meaningful to the exact situation, not all those needed by the most general case. For example, many of the C++ I/O functions make use of default arguments for just this reason.

A default argument can also be used as a flag telling the function to reuse a previous argument. To illustrate this usage, a function called **iputs( )** is developed here that automatically indents a string by a specified amount.

Program: Default argument

```
#include <iostream>
using namespace std;
/* Default indent to -1. This value tells the function
to reuse the previous value. */
void iputs(char *str, int indent = -1);
int main()
{
        iputs("Hello there", 10);
        iputs("This will be indented 10 spaces by default");
        iputs("This will be indented 5 spaces", 5);
        iputs("This is not indented", 0);
        return 0;
}
void iputs(char *str, int indent)
{
        static int i = 0; // holds previous indent value
        if(indent >= 0)
                i = indent;
        else // reuse old indent value
                indent = i;
        for( ; indent; indent--) cout << " ";
        cout << str << "\n";
}
```

Output:



```
          Hello there
          This will be indented 10 spaces by default
     This will be indented 5 spaces
This is not indented
```

When you are creating functions that have default arguments, it is important to remember that the default values must be specified only once, and this must be the first time the function is declared within the file.

All parameters that take default values must appear to the right of those that do not. For example, it is incorrect to define **iputs( )** like this:

```
// wrong!
void iputs(int indent = -1, char *str);
```

Once you begin to define parameters that take default values, you cannot specify a non defaulting parameter. That is, a declaration like this is also wrong and will not compile:

```
int myfunc(float f, char *str, int i=10, int j);
```

Program: Default arguments

```
#include <iostream>
using namespace std;

class cube
{
	int x, y, z;
	public:
	cube(int i=0, int j=0, int k=0)
	{
		x=i;
		y=j;
		z=k;
	}
	int volume()
	{
		return x*y*z;
	}
};

int main()
{
	cube a(2,3,4), b;
	cout << a.volume() << endl;
	cout << b.volume();
	return 0;
}
```

Advantages

There are two advantages to including default arguments,

1. First, they prevent you from having to provide an overloaded constructor that takes no parameters.

For example, if the parameters to **cube( )** were not given defaults, the second constructor shown here would be needed to handle the declaration of **b** (which specified no arguments).

```
cube() {x=0; y=0; z=0}
```

2. Second, defaulting common initial values is more convenient than specifying them each time an object is declared.

Inline Functions

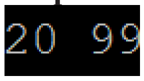
There is an important feature in C++, called an *inline function* that is commonly used with classes.

In C++, you can create short functions that are not actually called; rather, their code is expanded in line at the point of each invocation. This process is similar to using a function-like macro. To cause a function to be expanded in line rather than called, precede its definition with the

Program: Inline Function

```
#include <iostream>
using namespace std;

inline int max(int a, int b)
{
return a>b ? a : b;
}

int main()
{
cout << max(10, 20);
cout << " " << max(99, 88);
return 0;
}
```

Output:



```
20 99
```

The reason that **inline** functions are an important addition to C++ is that they allow you to create very efficient code. Since classes typically require several frequently executed interface functions (which provide access to private data), the efficiency of these functions is of critical concern. As you probably know, each time a function is called, a significant amount of overhead is generated by the calling and return mechanism. Typically, arguments are pushed onto the stack and various registers are saved when a function is called, and then restored when the function returns. The trouble is that these instructions take time. However, when a function is expanded in line, none of those operations occur. Although expanding function calls in line can produce faster run times, it can also result in larger code size because of duplicated code. For this reason, it is best to **inline** only very small functions. Further, it is also a good idea to **inline** only those functions that will have significant impact on the performance of your program.

Inline is actually just a *request*, not a command, to the compiler. The compiler can choose to ignore it.

Recursion

In C/C++, a function can call itself. A function is said to be *recursive* if a statement in the body of the function calls itself. Recursion is the process of defining something in terms of itself, and is sometimes called *circular definition*.
A simple example of a recursive function is **factr( )**,

```
/* recursive */
int factr(int n) {
int answer;
if(n==1) return(1);

answer = factr(n-1)*n; /* recursive call */
return(answer);
}
/* non-recursive */
int fact(int n) {
int t, answer;
answer = 1;
for(t=1; t<=n; t++)
answer=answer*(t);
return(answer);
}
```

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.

PROGRAM 59: RECURSION

```
#include <iostream>
#include<conio.h>
using namespace std;

int main()
{
long int factorial(int);
long int fact,value;
cout<<"Enter any number: ";
cin>>value;
fact=factorial(value);
cout<<"Factorial of a number is: "<<fact<<endl;
return 0;
}
long int factorial(int n)
{
if(n<0)
return(-1); /*Wrong value*/
if(n==0)
return(1);  /*Terminating condition*/
else
{
return(n*factorial(n-1));
}
}
```

OUTPUTS:

```
Enter any number: -1
Factorial of a number is: -1
Enter any number: 0
Factorial of a number is: 1
Enter any number: 17
Factorial of a number is: 355687428096000
```

Advantage

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.

Pointers to Functions

A particularly confusing yet powerful feature of C++ is the *function pointer*. Once a pointer points to a function, the function can be called through that pointer. Function pointers also allow functions to be passed as arguments to other functions.

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

PROGRAM 60: POINTERS TO FUNCTION

```
#include <iostream>
#include<conio.h>
#include <string.h>
using namespace std;

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;
cout<<"enter 2 strings";
gets(s1);
gets(s2);
check(s1, s2, p);
return 0;
}
void check(char *a, char *b,
int (*cmp)(const char *, const char *))
```

```
{
cout<<"Testing for equality.\n";
if(!(*cmp)(a, b)) cout<<"Equal";
else cout<<"Not Equal";
}
```

OUTPUT:

```
enter 2 strings hj
hj
Testing for equality.
Equal
```

STRINGS

C++ supports two types of strings.

a. Null-terminated string

Null-terminated string 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*.

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.

Null-terminated strings cannot be manipulated by any of the standard C++ operators. Nor can they take part in normal C++ expressions. For example, consider this fragment:

```
char s1[80], s2[80], s3[80];
s1 = "Alpha"; // can't do
s2 = "Beta"; // can't do
s3 = s1 + s2; // error, not allowed
```

As the comments show, in C++ it is not possible to use the assignment operator to give a character array a new value (except during initialization), nor is it possible to use the + operator to concatenate two strings. These

b. C++String Class

C++ also defines a string class, called **string**, which provides an object-oriented approach to string handling prevents such errors.

There are three reasons for the inclusion of the standard **string** class: consistency (a string now defines a data type), convenience (you may use the standard C++ operators), and safety (array boundaries will not be overrun).

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>**.)

PROGRAM 61: STRING TO READ A WORD

```
#include <iostream>
#include<conio.h>
#include <string.h>
using namespace std;

int main()
{
   char str[100];
   cout << "Enter a string: ";
   cin >> str;
   cout << "You entered: " << str << endl;
   cout << "\nEnter another string: ";
   cin >> str;
   cout << "You entered: "<<str<<endl;
   return 0;
}
```

OUTPUT:

```
Enter a string: KEERTHI
You entered: KEERTHI

Enter another string: KEERTHI
You entered: KEERTHI
```

PROGRAM 63: STRING USING STRING DATA TYPE

```
#include <iostream>
#include<conio.h>
#include <string.h>
using namespace std;

int main()
{
    // Declaring a string object
   string str;
   cout << "Enter a string: ";
   getline(cin, str);
   cout << "You entered: " << str << endl;
```

```
    return 0;
}
```

OUTPUT:

```
Enter a string: GOOD EVENING
You entered: GOOD EVENING
```

PROGRAM 64: STRING FUNCTIONS

```
#include<iostream>
using namespace std;
#include<conio.h>
#include <string.h>

int main ()
{
  char s1[80], s2[80];
  cout << "enter 2 strings";
  gets (s1);
  gets (s2);
  cout << "lengths: \n" << strlen (s1) << endl << strlen (s2) << endl;
  if (!strcmp (s1, s2))
    cout << "The strings are equal" << endl;
  strcat (s1, s2);
  cout << s1 << endl;
  strcpy (s1, "This is a test.\n");
  cout << s1;
  if (strchr ("hello", 'e'))
    cout << "e is in hello\n";
  if (strstr ("hi there", "hi"))
    cout << "found hi";
  return 0;
}
```

OUTPUT:

```
enter 2 stringshello
hello
lengths:
5
5
The strings are equal
hellohello
This is a test.
e is in hello
found hi
```

STRUCTURES

A structure is a collection of variables referenced under one name, providing a convenient means of keeping related information together. A *structure declaration* forms a template that may be used to create structure objects (that is, instances of a structure). The variables that make up the structure are called *members*. (Structure members are also commonly referred to as *elements* or *fields*.)

General form:

```
struct struct-type-name
{
        type member-name;
        type member-name;
        type member-name;
                ..
} structure-variables;
```

where either *struct-type-name* or *structure-variables* may be omitted, but not both.

Example:

```
struct addr
{
        char name[30];
        char street[40];
        char city[20];
        char state[3];
        unsigned long int zip;
} addr_info, binfo, cinfo;
```

At this point, *no variable has actually been created*. Only the form of the data has been defined. When you define a structure, you are defining a compound variable type, not a variable. Not until you declare a variable of that type does one actually exist. In C, to declare a variable (i.e., a physical object) of type **addr**, write

```
struct addr addr_info;
```

This declares a variable of type **addr** called **addr_info**. In C++, you may use this shorter form.

```
addr addr_info;
```

In C++, once a structure has been declared, you may declare variables of its type using only its type name, without preceding it with the keyword **struct.** The reason for this difference is that in C, a structure's name does not define a complete type name. In fact, Standard C refers to a structure's name as a *tag*. In C, you must precede the tag with the keyword **struct** when declaring variables. However, in C++, a structure's name is a complete type name and may be used by itself to define variables.

When a structure variable (such as **addr_info**) is declared, the compiler automatically allocates sufficient memory to accommodate all of its members.

Accessing Structure Members

Individual members of a structure are accessed through the use of the **.** operator (usually called the *dot operator*).

The structure variable name followed by a period and the member name references that individual member. The general form for accessing a member of a structure is

structure-name.member-name

Therefore, to print the ZIP code on the screen, write

```
printf("%lu", addr_info.zip);
```

Structure Assignments

The information contained in one structure may be assigned to another structure of the same type using a single assignment statement. That is, you do not need to assign the value of each member separately.

```
struct {
int a;
int b;
} x, y;
x.a = 10;
y = x; /* assign one structure to another */
```

PROGRAM 66: STRUCTURE ASSIGNMENT

```
#include <iostream>
using namespace std;

int main()
{
   struct {
int a;
int b;
} x, y;
x.a = 10;
y = x; /* assign one structure to another */
cout<<y.a;
return 0;
}
```

OUTPUT:

```
10
```

Arrays of Structures

To declare an array of structures, you must first define a structure and then declare an array variable of that type. For example, to declare a 100-element array of structures of type **addr**, defined earlier, write

```
struct addr addr_info[100];
```

To access a specific structure, index the structure name. For example, to print the ZIP code of structure 3, write

```
printf("%lu", addr_info[2].zip);
```

Passing Structures to Functions

a. Passing Structure Members to Functions

When you pass a member of a structure to a function, you are actually passing the value of that member to the function. Therefore, you are passing a simple variable

For example,
consider this structure:

```
struct fred
{
char x;
int y;
float z;
char s[10];
} mike;
```

Here are examples of each member being passed to a function:

```
func(mike.x); /* passes character value of x */
```

```
func2(mike.y); /* passes integer value of y */
func3(mike.z); /* passes float value of z */
func4(mike.s); /* passes address of string s */
func(mike.s[2]); /* passes character value of s[2] */
```

If you wish to pass the *address* of an individual structure member, put the **&** operator before the structure name. For example, to pass the address of the members of the structure **mike**, write

```
func(&mike.x); /* passes address of character x */
func2(&mike.y); /* passes address of integer y */
func3(&mike.z); /* passes address of float z */
func4(mike.s); /* passes address of string s */
func(&mike.s[2]); /* passes address of character s[2] */
```

b. Passing Entire Structures to Functions

When a structure is used as an argument to a function, the entire structure is passed using the standard call-by-value method. Of course, this means that any changes made to the contents of the structure inside the function to which it is passed do not affect the structure used as an argument.

```
/* Define a structure type. */
struct struct_type {
int a, b;
char ch;
} ;
void f1(struct struct_type parm);
int main(void)
{
struct struct_type arg;
arg.a = 1000;
f1(arg);
return 0;
}
void f1(struct struct_type parm)
{
printf("%d", parm.a);
}
```

As this program illustrates, if you will be declaring parameters that are structures, you must make the declaration of the structure type global so that all parts of your program can use it. For example, had **struct_type** been declared inside **main( )** (for example), then it would not have been visible to **f1( ).**

PROGRAM 67: PASSING A STRUCTURE TO A FUNCTION

```
#include <iostream>
using namespace std;

struct struct_type
{
int a, b;
char ch;
} ;
void f1(struct struct_type parm);
int main(void)
{
struct struct_type arg;
arg.a = 1000;
f1(arg);
return 0;
}
void f1(struct struct_type parm)
{
cout<<parm.a;
}
```

OUTPUT:

```
1000
```

PROGRAM 68: PASSING A STRUCTURE TO A FUNCTION

```
#include <iostream>
using namespace std;

struct Person
{
   char name[50];
   int age;
   float salary;
};
void displayData(Person);   // Function declaration
int main()
{
   Person p;
   cout << "Enter Full name: ";
   cin.get(p.name, 50);
   cout << "Enter age: ";
   cin >> p.age;
   cout << "Enter salary: ";
   cin >> p.salary;
   // Function call with structure variable as an argument
   displayData(p);
   return 0;
}
void displayData(Person p)
{
   cout << "\nDisplaying Information." << endl;
   cout << "Name: " << p.name << endl;
   cout <<"Age: " << p.age << endl;
   cout << "Salary: " << p.salary;
}
```

OUTPUT:

```
Enter Full name: ABC
Enter age: 23
Enter salary: 25000

Displaying Information.
Name: ABC
Age: 23
Salary: 25000
```

Structure Pointers

C++ allows pointers to structures just as it allows pointers to any other type of variable.

Declaring a Structure Pointer

Like other pointers, structure pointers are declared by placing ***** in front of a structure variable's name. For example, assuming the previously defined structure **addr**, the following declares **addr_pointer** as a pointer to data of that type:

```
struct addr *addr_pointer;
```

Remember, in C++ it is not necessary to precede this declaration with the keyword **struct**.

Using Structure Pointers

There are two primary uses for structure pointers: to pass a structure to a function using call by reference, and to create linked lists and other dynamic data structures that rely on dynamic allocation.

When a pointer to a structure is passed to a function, only the address of the structure is pushed on the stack. This makes for very fast function calls. A second advantage, in some cases, is when a function needs to reference the actual structure used as the argument, instead of a copy. By passing a pointer, the function can modify the contents of the structure used in the call.

To find the address of a structure, place the **&** operator before the structure's name. For example, given the following fragment:

```
struct bal {
float balance;
char name[80];
} person;
struct bal *p; /* declare a structure pointer */
```

then

```
p = &person;
```

places the address of the structure **person** into the pointer **p**.

To access the members of a structure using a pointer to that structure, you must use the -> operator. For example, this references the **balance** field:

```
p->balance
```

The -> is usually called the *arrow operator*, and consists of the minus sign followed by a greater-than sign. The arrow is used in place of the dot operator when you are accessing a structure member through a pointer to the structure.

PROGRAM 69: POINTER STRUCTURES

```
#include <iostream>
```

```
using namespace std;

struct Distance
{
   int feet;
   float inch;
};
int main()
{
   Distance *ptr, d;
   ptr = &d;

   cout << "Enter feet: ";
   cin >> (*ptr).feet;
   cout << "Enter inch: ";
   cin >> (*ptr).inch;

   cout << "Displaying information." << endl;
   cout << "Distance = " << (*ptr).feet << " feet " << (*ptr).inch << " inches";
   return 0;
}
```

OUTPUT:

```
Enter feet: 10
Enter inch: 2
Displaying information.
Distance = 10 feet 2 inches
```

REFERENCES

C++ contains a feature that is related to the pointer called a *reference*. A reference is essentially an implicit pointer. There are three ways that a reference can be used: as a function parameter, as a function return value, or as a stand-alone reference.

Reference Parameters

Probably the most important use for a reference is to allow you to create functions that automatically use call-by-reference parameter passing. Arguments can be passed to functions in one of two ways: using call-by-value or call-by-reference. When using call-by-value, a copy of the argument is passed to the function. Call-by-reference passes the address of the argument to the function

By default, C++ uses call-by-value, but it provides two ways to achieve call-by-reference parameter passing. First, you can explicitly pass a pointer to the argument. Second, you can use a reference parameter. For most circumstances the best way is to use a reference parameter.

PROGRAM 70: REFERENCES
WITHOUT REFERENCE

```
// Manually create a call-by-reference using a pointer.
```

```
#include <iostream>
using namespace std;

void neg(int *i);
int main()
{
int x;
x = 10;
cout << x << " negated is ";
neg(&x);
cout << x << "\n";
return 0;
}
void neg(int *i)
{
*i = -*i;
}
```

WITH REFERENCE

```
#include <iostream>
using namespace std;

void neg(int &i); // i now a reference
int main()
{
int x;
x = 10;
cout << x << " negated is ";
neg(x); // no longer need the & operator
cout << x << "\n";
return 0;
}
void neg(int &i)
{
i = -i; // i is now a reference, don't need *
}
```

OUTPUT:

```
10 negated is -10
```

Returning References

A function may return a reference

simple program:

```
#include <iostream>
using namespace std;
char &replace(int i); // return a reference
char s[80] = "Hello There";
int main()
{
replace(5) = 'X'; // assign X to space after Hello
cout << s;
return 0;
}
char &replace(int i)
{
return s[i];
}
```

One thing you must be careful about when returning references is that the object being referred to does not go out of scope after the function terminates.

PROGRAM 72: RETURNING REFERENCES

```
#include <iostream>
using namespace std;

char &replace(int i); // return a reference
char s[80] = "Hello There";
int main()
{
replace(5) = 'X'; // assign X to space after Hello
cout << s;
return 0;
}
char &replace(int i)
{
return s[i];
}
```

OUTPUT:

```
HelloXThere
```

C++'S DYNAMIC ALLOCATION OPERATORS

C++ provides two dynamic allocation operators: **new** and **delete**. C++ also supports dynamic memory allocation functions, called **malloc( )** and **free( ).** These are included for the sake of compatibility with C.

The **new** operator allocates memory and returns a pointer to the start of it. The **delete** operator frees memory previously allocated using **new**.

General forms:

p_var = new *type*;
delete *p_var*;

Here, *p_var* is a pointer variable that receives a pointer to memory that is large enough to hold an item of type *type*.

Since the heap is finite, it can become exhausted. If there is insufficient available memory to fill an allocation request, then **new** will fail and a **bad_alloc** exception will be generated. This exception is defined in the header **<new>**. Your program should handle this exception and take appropriate action if a failure occurs

The **delete** operator must be used only with a valid pointer previously allocated by using **new**. Using any other type of pointer with **delete** is undefined and will almost certainly cause serious problems, such as a system crash.

Although **new** and **delete** perform functions similar to **malloc( )** and **free( )**, they have several advantages.

1. First, **new** automatically allocates enough memory to hold an object of the specified type. You do not need to use the **sizeof** operator. Because the size is computed automatically, it eliminates any possibility for error in this regard.
2. Second, **new** automatically returns a pointer of the specified type. You don't need to use an explicit type cast as you do when allocating memory by using **malloc( )**. Finally, both **new** and **delete** can be overloaded, allowing you to create customized allocation systems.

Initializing Allocated Memory

You can initialize allocated memory to some known value by putting an initialize after the type name in the **new** statement. Here is the general form of **new** when an initialization is included:

General Form*:* *p_var* = new *var_type (initializer);*

PROGRAM 73: DYNAMIC MEMORY ALLOCATION AND DEALLOCATION TO HOLD AN INTEGER

```
#include <iostream>
using namespace std;

int main()
{
        int *p;
        try {
                p = new int; // allocate space for an int
        } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;
}
        *p = 100;
        cout << "At " << p << " ";
        cout << "is the value " << *p << "\n";
        delete p;
        return 0;
}
```

OUTPUT:

```
At 0x162cc20 is the value 100
```

PROGRAM 74: DYNAMIC MEMORY ALLOCATION AND DEALLOCATION WHICH GIVES THE ALLOCATED INTEGER AN INITIAL VALUE

```
#include <iostream>

using namespace std;

int main()
{
        int *p;
        try {
        p = new int (87); // initialize to 87
        } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;
}
        cout << "At " << p << " ";
        cout << "is the value " << *p << "\n";
        delete p;
        return 0;
}
```

OUTPUT:

```
At 0xc14c20 is the value 87
```

Allocating Arrays

You can allocate arrays using **new.**

General form: *p_var* = new *array_type [size];*

Here, *size* specifies the number of elements in the array.

To free an array, use this form of **delete:**

General form: delete [ ] *p_var*;

Here, the **[ ]** informs **delete** that an array is being released.

PROGRAM: DYNAMIC MEMORY ALLOCATION AND DEALLOCATION OF ARRAYS

```
#include <iostream>
using namespace std;

int main()
{
        int *p, i;
        try {
        p = new int [10]; // allocate 10 integer array
        } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;
        }
        for(i=0; i<10; i++ )
        p[i] = i;
        for(i=0; i<10; i++)
        cout << p[i] << " ";
        delete [] p; // release the array
        return 0;
}
```

OUTPUT:

```
0 1 2 3 4 5 6 7 8 9
```

The Placement Form of new

There is a special form of **new**, called the *placement form*, that can be used to specify an alternative method of allocating memory. It is primarily useful when overloading the **new** operator for special circumstances.

General form: *p_var* = new (*arg-list*) *type*;

Here, *arg-list* is a comma-separated list of values passed to an overloaded form of **new**.

PREPROCESSOR DIRECTIVES

You can include various instructions to the compiler in the source code of a C/C++ program. These are called *preprocessor directives*

C++ preprocessor is virtually identical to the one defined by C. The main difference between C and C++ in this regard is the degree to which each relies upon the preprocessor. In C, each preprocessor directive is necessary. In C++, some features have been rendered redundant by newer and better C++ language elements.

All preprocessor directives begin with a # sign. In addition, each preprocessing directive must be on its own line.

For example,

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

will not work.

The preprocessor contains the following directives:

#define

The **#define** directive defines an identifier and a character sequence (i.e., a set of characters) that will be substituted for the identifier each time it is encountered in the source file. The identifier is referred to as a *macro name* and the replacement process as *macro replacement*.

General form : #define *macro-name char-sequence*

Notice that there is no semicolon in this statement.

Example:

```
#define LEFT 1
#define RIGHT 0
```

#if , #elif , #else , #endif

Perhaps the most commonly used conditional compilation directives are the **#if**, **#else**, **#elif**, and **#endif**. These directives allow you to conditionally include portions of code based upon the outcome of a constant expression.

General form of #if :

#if *constant-expression*
statement sequence
#endif

If the constant expression following **#if** is true, the code that is between it and **#endif** is compiled. Otherwise, the intervening code is skipped. The **#endif** directive marks the end of an **#if** block.

For example,

```
/* Simple #if example. */
#include <stdio.h>
#define MAX 100
int main(void)
{
#if MAX>99
cout<<"Compiled for array greater than 99.\n";
#endif
return 0;
```

}
This program displays the message on the screen because **MAX** is greater than 99.

The **#elif** directive means "else if" and establishes an if-else-if chain for multiple compilation options. **#elif** is followed by a constant expression. If the expression is true, that block of code is compiled and no other **#elif** expressions are tested. Otherwise, the next block in the series is checked.

General form for #elif :

```
#if expression
        statement sequence
#elif expression 1
        statement sequence
#elif expression 2
        statement sequence
#elif expression 3
        statement sequence
#elif expression 4
.
.
.
#elif expression N
        statement sequence
#endif
```

#error

The **#error** directive forces the compiler to stop compilation. It is used primarily for debugging.

General form : #error *error-message*

The *error-message* is not between double quotes. When the **#error** directive is encountered, the error message is displayed, possibly along with other information defined by the compiler.

#ifdef , #ifndef

Another method of conditional compilation uses the directives **#ifdef** and **#ifndef**, which mean "if defined" and "if not defined," respectively.

General form of #ifdef:

```
#ifdef macro-name
        statement sequence
#endif
```

If *macro-name* has been previously defined in a **#define** statement, the block of code will be compiled.

General form of #ifndef:

```
#ifndef macro-name
        statement sequence
#endif
```

If *macro-name* is currently undefined by a **#define** statement, the block of code is compiled.

Both **#ifdef** and **#ifndef** may use an **#else** or **#elif** statement.

For example,

```
#include <stdio.h>
#define TED 10
int main(void)
{
#ifdef TED
cout<<"Hi Ted\n";
#else
cout<<"Hi anyone\n";
#endif
#ifndef RALPH
cout<<"RALPH not defined\n";
#endif
return 0;
}
```

You may nest **#ifdef**s and **#ifndef**s to at least eight levels in Standard C. Standard C++ suggests that at least 256 levels of nesting be supported.

#include

The **#include** directive instructs the compiler to read another source file in addition to the one that contains the **#include** directive. The name of the additional source file must be enclosed between double quotes or angle brackets.

For example,

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

both instruct the compiler to read and compile the header for the C I/O system library functions.

Include files can have **#include** directives in them. This is referred to as *nested includes*. The number of levels of nesting allowed varies between compilers. However, Standard C stipulates that at least eight nested inclusions will be available. Standard C++ recommends that at least 256 levels of nesting be supported.

#undef

The **#undef** directive removes a previously defined definition of the macro name that follows it. That is, it "undefines" a macro.

General form : #undef *macro-name*

For example,

```
#define LEN 100
#define WIDTH 100
char array[LEN][WIDTH];
#undef LEN
#undef WIDTH
/* at this point both LEN and WIDTH are undefined */
```

Both **LEN** and **WIDTH** are defined until the **#undef** statements are encountered. **#undef** is used principally to allow macro names to be localized to only those sections of code that need them.

#line

The **#line** directive changes the contents of **_ _LINE_ _** and **_ _FILE_ _ ,** which are predefined identifiers in the compiler. The **_ _LINE_ _** identifier contains the line number of the currently compiled line of code. The **_ _FILE_ _** identifier is a string that contains the name of the source file being compiled.

General form: #line *number* "*filename*"

where *number* is any positive integer and becomes the new value of **_ _LINE_ _ ,** and the optional *filename* is any valid file identifier, which becomes the new value of **_ _FILE_ _. #line** is primarily used for debugging and special applications.

For example, the following code specifies that the line count will begin with 100. The cout statement displays the number 102 because it is the third line in the program after the **#line 100** statement.

For example, the following code specifies that the line count will begin with 100. The cout statement displays the number 102 because it is the third line in the program after the **#line 100** statement.

```
#include <stdio.h>
#line 100 /* reset the line counter */
int main(void) /* line 100 */
{ /* line 101 */
cout<<__LINE__; /* line 102 */
return 0;
}
```

#pragma

#pragma is an implementation-defined directive that allows various instructions to be given to the compiler. For example, a compiler may have an option that supports program execution tracing. A trace option would then be specified by a **#pragma** statement.

PROGRAM: PREPROCESSOR DIRECTIVES (#define)

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

using namespace std;

#define PI 3.14159

int main () {
  cout << "Value of PI :" << PI << endl;

  return 0;
}
```

OUTPUT:

```
Value of PI :3.14159
```