**Tokens**

The smallest individual units in a program are known as tokens. C++ has the following tokens:

1. Keywords
2. Identifiers
3. Constants
4. Strings
5. Operators

C++program is written using these tokens, white spaces, and the syntax of the language

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

1. **Keywords**

Keywords are reserved identifiers and cannot be used as names for the program variables or other user defined program.

Ex: double, new, switch, auto, else, break etc.



1. **Identifiers**

Identifiers refer to the names of variables, functions, arrays, classes, etc. created by the programmer.

**The following rules are common to both C and C++:**

* Only alphabetic characters, digits and underscores are permitted.
* The name cannot start with a digit.
* Uppercase and lowercase letters are distinct.
* A declared keyword cannot be used as a variable name.

1. **Constants:**

Constants refer to fixed values that do not change during the execution of a program. C++ supports several kinds of literal constants. They include **integers, characters, floating point numbers and strings**. Literal constant do not have memory locations. Examples:

123 -- Decimal integer

12.34 -- floating point integer

037 -- Octal integer

OX2 -- hexadecimal integer

"C++" -- string constant

'A' -- character constant

L'ab' -- wide-character constant

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**2) Basic or (fundamental) Data Types**





**Void:** Two normal uses of void are **(1) to specify the return type of a function when it is not returning any value (2) to indicate an empty argument list to a function.**

Example: void functl(void);

**Generic pointers**: Another interesting use of void is in the declaration of generic pointers.

Example: void \*gp; // gp becomes generic pointer

A generic pointer can be assigned a pointer value of any basic data type, but it may not be dereferenced.

Example: int \*ip; // int pointer

gp = ip; //assign int pointer to void pointer

are valid statements. But, the statement,

\*ip = \*gp; is illegal. It would not make sense to dereference a pointer to a void value.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. **User-Defined or (primitive) Data Types**
2. **Structures :** structures are used for grouping together elements with different data types.

**The general format:**

struct name

{

data type memberl;

data type member2;

............................

} ;

Let us take the example of a book, which has several attributes such as title, number of pages, price, etc.

struct book

{

char title[25);

char author[25);

int pages;

float price;

} ;

struct book bookl, book2, book3;

Here book1, book2 and book3 are declared as variables of the user-defined type book. We can access the member elements of a structure by using the dot (.) operator, as shown below:

bookl.pages=550;

book2.price=225.75;

1. **UNIONS :** Unions are conceptually similar to structures as they allow us to group together dissimilar type elements inside a single unit. **The size of a structure type is equal to the sum of the sizes of individual member types. However, the size of a union is equal to the size of its largest member element.**

union result

{

int marks;

char grade;

float percent;

} ;

**The union result will occupy four bytes in memory as its largest size member element is the floating type variable percent**. However, if we had defined result as a **structure then it would have occupied seven bytes in memory that is, the sum of the sizes of individual member elements.** Thus, in case of unions the same memory space is used for representing different member elements. As a result, union members can only be manipulated exclusive of each other. In simple words, we can say that unions are memory-efficient alternatives of structures particularly in situations where it is not required to access the different member elements simultaneously.

****

1. **Enumerated Data Type**

An enumerated data type is another user-defined type which provides a way for **attaching names to numbers**, thereby increasing comprehensibility of the code. The enum keyword automatically enumerates a list of words by assigning those values **0, 1, 2,** and so on. This facility provides an alternative means for creating symbolic constants. The syntax of an enum statement is similar to that of the struct statement.

Examples: enum shape{circle, square, triangle};

enum colour{red, blue, green, yellow};

enum position{off, on};

In C++, the tag names shape, colour, and position become new type names. By using these tag names, we can declare new variables.

Examples: shape ellipse; // ellipse is of type shape

colour background; // background is of type colour

By default, the enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second, and so on. We can over-ride the default by explicitly assigning integer values to the enumerators.

Example: enum colour{red, blue=4, green=8};

enum colour{red=5, blue, green};

are valid definitions. In the first case, red is 0 by default. In the second case, blue is 6 and green is 7.

**Example :**

enum shape

{

circ1e,

rectangle,

triangle

} ;

int main ()

{

cout « "Enter shape code:";

int code;

cin » code;

switch(code)

{

case circle: -----

break;

case rectangle: -----

break;

case triangle: -----

break;

default : -----

}

cout « "BYE \n";

return 0;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. **Derived Data Types**
2. **Arrays:**

The application of arrays in C++ is similar to that in C. The only exception is the way character arrays are initialized. When initializing a character array in ANSI C, the compiler will allow us to declare the array size as the exact length of the string constant. For instance,

char string[3] = "xyz";

is valid in ANSI C. It assumes that the programmer intends to leave out the null character \0 in the definition. But in C++, the size should be one larger than the number of characters in the string.

char string[4] ="xyz"; / / O. K. for C++

1. **Pointers**

Pointers are variable which hold the address of another variable.

int \*ip; // int pointer

ip = &x; // address of x assigned to ip

\*ip = 10 // 10 assigned to x through indirection

C++ adds the concept of constant pointer and pointer to a constant.

char \* const ptr1 = “GOOD”; // constant pointer

We cannot modify the address that ptr1 is initialized to.

int const \* ptr2 = &m; // pointer to a constant

**ptr2** is declared as pointer to a constant. It can point to any variable of correct type, but the contents of what it points to cannot be changed.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. **Symbolic Constants**

There are two ways of creating symbolic constants in C++:

* Using the qualifier **const**
* Defining a set of integer constants using **enum** keyword.

In C++, any value declared as **const** cannot be modified by the program in any way. In C++, we can use **const** in a constant expression, such as

const int size = 10;

char name[size];

If we use the const modifier alone, it defaults to in to for example, const size = 10; means const int size = 10;

Another method of naming integer constants is by enumeration as under;

enum {X,Y,Z};

This defines X, Y and Z as integer constants with values 0, 1, and 2 respectively. This is

equivalent to:

const X = 0;

const Y = 1;

const Z = 2;

We can also assign values to X, Y, and Z explicitly.

Example: enum{X=100,Y=50, Z=200};

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. **Declaration of Variables**

C++ allows the declaration of a variable anywhere in the scope. This means that a variable can be declared right at the place of its first use. This makes the program much easier to write and reduces the errors that may be caused by having to scan back and forth. It also makes the program easier to understand because the variables are declared in the context of their use.

**Example :**

int main()

{

Float x; **// declaration**

Float sum=0; **// declaration**

For(int i=1;i<5;i++) **// declaration**

{

cin>>x;

Sum=sum+x;

}

Float average; **//declaration**

Average=sum/(i-1);

Cout<<average;

Return 0;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. **Dynamic Initialization of Variables**

In C++, permits initialization of the variables at run time referred to as dynamic initialization. In C++, a variable can **be initialized at run time using expressions at the place of declaration.**

Example: int n = strlen(string)j

float area = 3.14159 \* rad \* rad;

Thus, both the declaration and the initialization of a variable can be done simultaneously at the place where the variable is used for the first time. **The following two statements in the example of the previous section**

float average; // declare where it is necessary

average = sum/i;

**can be combined into a single statement:**

float average = sum/i; //initialize Dynamically at run time

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. **Reference Variables**

A reference variable provides an alias (alternative name) for a previously defined variable. For example, if we make the variable sum a reference to the variable total, then sum and total can be used interchangeably to represent that variable. A reference variable is created as follows:

Syntax : **data-type & reference-name = variable-name**

Example: float total = 100;

float & sum = total;

total is a float type variable that has already been declared; sum is the alternative name declared to represent the variable total. Both the variables refer to the same data object in the memory. Now, the statements

cout «total;

and

cout «sum; both print the value 100.

The statement total = total + 10; Will change the value of both total and sum to 110.

The assignment sum = 0; will change the value of both the variables to zero.

A reference variable must be initialized at the time of declaration. This establishes the correspondence between the reference and the data object which it names. It is important to note that the initialization of a reference variable is completely different from assignment to it.

C++ assigns additional meaning to the symbol &. Here, & is not an address operator. The notation float & means reference to float. Other examples are:

int n[10];

int & x = n[10]; // x is alias for n[10]

The variable x is an alternative to the array element n[l0].

The following references are also allowed:

1. int x;

int \*p = &x;

int & m = \*p;

1. int & n = 50;

The first set of declarations causes m to refer to x which is pointed to by the pointer p and the statement in (ii) creates an int object with value 50 and name n.

A major application of reference variables is in passing arguments to functions. Consider the following:

void f(int & x) // uses reference

{

x = x+10; // x is incremented; so also m

}

int main0

{

int m=10;

**f(m);**  // function call

......

}

When the function call f(m) is executed, the following initialization occurs:

int & x = m;

Thus x becomes an alias of m after executing the statement

**f(m); calls are known as call by reference.**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. **Operators in C+ +**

C++ has a rich set of operators. All C operators are valid in C++ also. In addition, C++ introduces some new operators. We have already seen two such operators, namely, the insertion operator <<, and the extraction operator >>. Other new operators are:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | :: | Scope resolution operator | | ::\* | Pointer-to-member declarator | | -> | Pointer-to-member operator | | .\* | Pointer-to-member operator | | delete | Memory release operator | | endl | Line feed operator | | new | Memory allocation operator | | setw | Field width operator | |

1. **Scope resolution operator: (::)**

In C, the global version of a variable cannot be accessed from within the inner block. C++ resolves this problem by introducing a new operator **::** called the scope resolution operator. This can be used to uncover a hidden variable. It takes the following form:

**:: variable-name**

This operator allows access to the global version of a variable.

Example ::count means the global version of the variable count (and not the local variable count declared in that block)



**Out put:**

We are in inner block

k = 20

m = 30

::m = 10

We are in outer block

m = 20

::m = 10

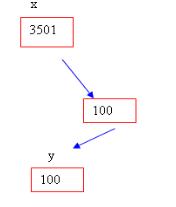
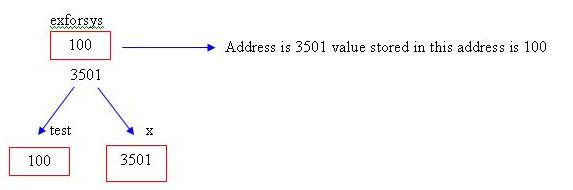
1. **Member Dereferencing Operators**

C++ permits us to define a class containing various types of data and functions as members. C++ also permits us to access the class members through pointers. In order achieve this, C++ provides a set of three pointer-to-member operators.

|  |  |
| --- | --- |
| Operator | Function |
| ::\* | declare a pointer to a member of a class |
| \* | To access a member using object name and a pointer to that member |
| ->\* | To access a member using a pointer to the object and a pointer to that member |

**For example:**

* **exforsys = 100;**
* **test = exforsys;**
* **x = &exforsys;**
* **y=\*x; // dereferencing operator (This means that the value pointed to by the pointer variable x gives the value 100 to y.)**

****

1. **Memory Management Operators**

C uses malloc() and calloc() functions to allocate memory dynamically at run time. C++ uses the function free () to free dynamically allocated memory. We use dynamic allocation techniques when it is not known in advance how much of memory space is needed. C++ supports these functions; it also defines two unary operators **new** and **delete** that perform the task of allocating and freeing the memory in a better and easier way.

An object can be created by using new, and destroyed by using delete, as and when required. A data object created inside a block with new, will remain in existence until it is explicitly destroyed by using delete.

**New operator :**The new operator can be used to create objects of any type. It takes the following general form:

**Pointer-variable = new data-type;**

Here, pointer-variable is a pointer of type data-type. The new operator allocates sufficient memory to hold a data object of type data-type and returns the address of the object. The data- type may be any valid data type. The pointer-variable holds the address of the memory space allocated.

Examples: p = new int;

q = new float;

Where p is a pointer of type int and q is a pointer of type float. Here, p and q must have already been declared as pointers of appropriate types. Alternatively, we can combine the declaration of pointers and their assignments as follows:

int \*p = new int;

float \*q = new float;

Subsequently, the statements

\*p = 25;

\*q = 7.5;

Assign 25 to the newly created int object and 7.5 to the float object.

We can also initialize the memory using the **new** operator. This is done as follows:

**pointer-variable = new data-type(value);**

Here, value specifies the initial value.

Examples: int \*p = new int(25);

float \*q = new float(7.5);

new can be used to create a memory space for any data type including user-defined types such as arrays, structures and classes. The general form of one-dimensional array is:

**pointer-variable new data-type[size];**

Here, size specifies the number of elements in the array. For example, the statement

int \*p = new int[10];

**delete operator :**

When a data object is no longer needed, it is destroyed to release the memory space for reuse. The general form of its use is:

**delete pointer-variable;**

The pointer-variable is the pointer that points to a data object created with new. Examples: delete p;

delete q;

If we want to free a dynamically allocated array, we must use the following form of delete:

**delete [size] pointer-variable;**

The size specifies the number of elements in the array to be freed. The problem with this form is that the programmer should remember the size of the array. Recent versions of C++ do not require the size to be specified. For example,

**delete [ ] p;**

will delete the entire array pointed to by p.

**The new operator offers the following advantages over the function malloc()**

1. It automatically computes the size of the data object. We need not use the operator sizeof.
2. It automatically returns the correct pointer type, so that there is no need to use a type cast.
3. It is possible to initialize the object while creating the memory space.
4. Like any other operator, new and delete can be overloaded.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

1. **Manipulators**

Manipulators are operators used with the insertion operator << to modify or manipulate the way data is displayed. Two most common manipulators are endl and setw.

**endl**

This is a manipulator that causes a linefeed to be inserted into the stream. It has the same effect as sending the single '\n' character, but is perhaps somewhat clearer.

Eg: #include<iostream.~h

void main( )

{

int a=10,b=20,c=30;

cout << “Value of a ="<<a<<end1;

cout << “Value of b ="<<b<<end1;

cout << “Value of c ="<<c<<end1;

}

**Output :**

Value of a=10

Value of b=20

Value of c=30

**setw()**

The value displayed by cout occupies a field: an imaginary box with a certain width. The default field is just wide enough to hold the value. However, in certain situations this may lead to optimal results. The setw manipulator causes the member (or string) that follows it in the stream to be printed within a field n characters wide, where n is the argument to setw(n). The value is right justified within the field.

Eg: #include<iostream.h>

void main()

{

long pop1=2425785; pop2=47;

cout<<setw(8)<<"LOCATION"<<setw(12)<<"POPULATION"<<endl;

cout<<setw(8)<<"KARKALA"<<setw(12)<< pop1<<endl;

cout<<setw(8)<<"UDUPI”<<setw(12)<<pop2<<endl;

}

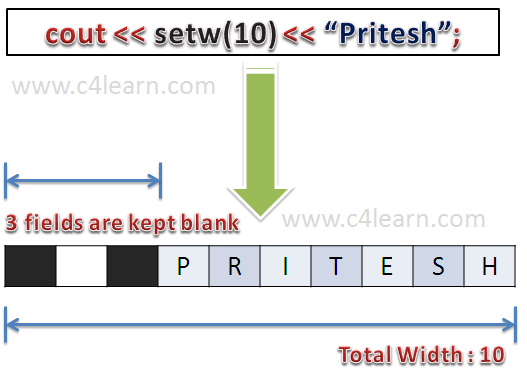
Out put:

LOCATION POPULATION

KARKALA 2425785

UDUPI 47

**( Or )**

****

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

1. **Type Cast Operator**

C++ permits explicit type conversion of variables or expressions using the type cast operator.

**type-name (expression)**

Example: sum/float(i);

**A type-name behaves as if it is a function for converting values to a designated type.**

Eg: #include<iostream.h>

void main()

{

int sum=45,000;

int n=55;

float avg;

avg=(float)sum/n;

}

In above example when we divide an integer variable sum by another integer n, then the result will be an integer. The fractional part will be truncated. And since avg is a float variable to which the result is to be assigned, the result will be a float but fractional part is truncated.(Ex: 56.000000)

1. **Expressions and Their Types**

An expression is a combination of operators, constants and variables arranged as per the rules of the language. It may also include function calls which return values. An expression may consist of one or more operands, and zero or more operators to produce a value. **Expressions may be of the following seven types:**

1. Constant expressions
2. Integral expressions
3. Float expressions
4. Pointer expressions
5. Relational expressions
6. Logical expressions
7. Bitwise expressions
8. **Constant Expressions:**

Constant Expressions consist of only constant values.

Examples: 15

20 + 5 / 2.0

‘x’

1. **Integral Expressions:**

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

Examples: m

m \* n - 5

1. **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

**d.) Pointer Expressions:**

Pointer Expressions produce address values.

Examples: &m

Ptr

1. **Relational Expressions or Boolean expression:**

Relational Expressions yield results of type bool which takes a value true or false. Examples: x <= y

a+b c+d

m+n > 100

**f) Logical Expressions:**

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

Examples:

a>b && x==10

x==10 || y==5

**g) 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

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**13) Special Assignment Expressions**

**a) 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.

**b) 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;

1. **Compound Assignment**

This 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 assignmnent operator.

Variable1 operator=variable2;

Where op is binary arithmetic operator. This means that.

Variable1 =variable1 op variable2;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**14) Implicit Conversions( done automatically by program itself).**

When two operands of different types are encountered in, the same expression, the lower-type variable is converted to the type of higher-type variable.

These conversions take place invisibly, C++ automatically does it. However when we start to use objects, we must be careful to create our own conversion routines to change objects of one type into objects of another. The compiler won't do it for us, as it does with the built in data types.

Ex: int a=5;

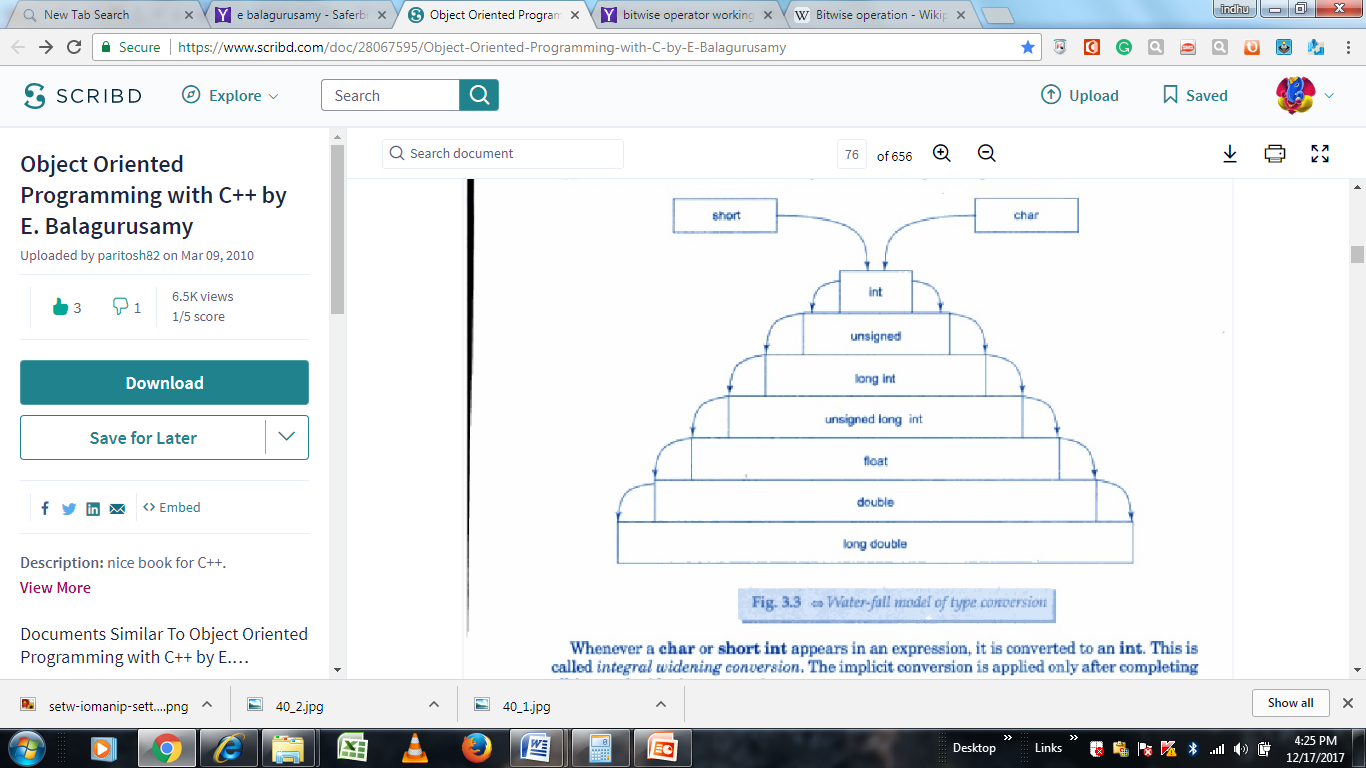
Float b;

b=a;

cout<<b;

Here the value of b=5.000000

For a binary operator, if the operands type differ, the compiler converts one of them to match with the other, using the rule that the "smaller" type is converted to the "wider" type. For example, if one of the operand is an int and the other is a float, the int is converted into a float because a float is wider than an int.



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

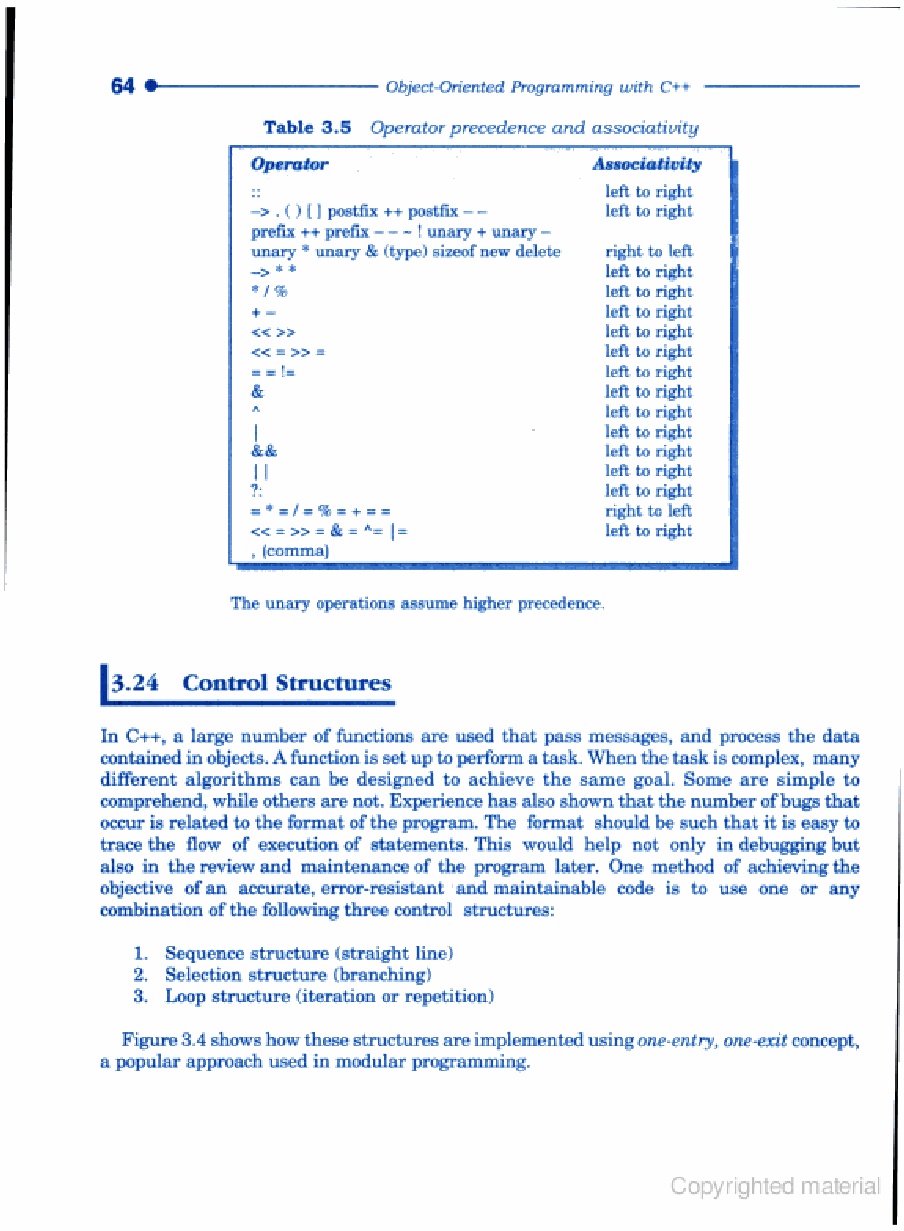
**15) Operator Precedence**

Although C++ enables us to add multiple meanings to the operators, yet their association and precedence remain the same. For example, the multiplication operator will continue having higher precedence than the add operator.

() [] // higher precedence

\* / %

+ - //lower precedence

****

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

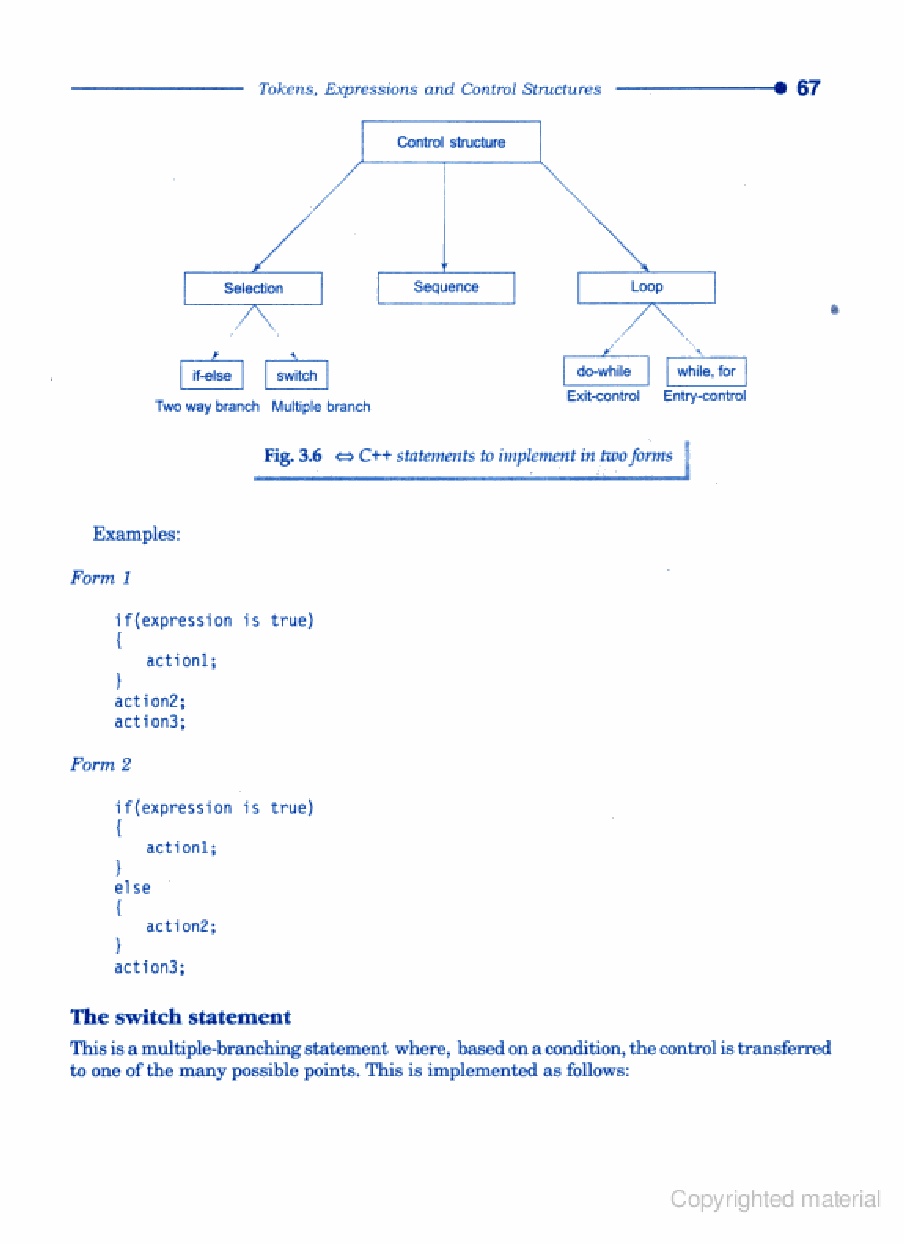
**16) Control Structures**

1. Sequence structure (straight line)
2. Selection structure (branching)
3. Loop structure (iteration or repetition)



It is important to understand that all program processing can be coded using these 3 logics.

The approach of using one or more of these basic control constructs in programming is known as “STRUCTURED PROGRAMMING”.



**if statement**

The if statement is implemented in two forms:

1. **Simple if statement**
2. **If...else statement**
3. **Simple if statement**

if(expression is true)

{

Action 1;

}

action2;

action3;

1. **if ...else statement**

if(expression(is true)

{

Action 1:

}

else

{

Action2;

}

action3;

|  |  |  |  |
| --- | --- | --- | --- |
| **sequence structure** | **Example flowchart** | **Step by step instructions** |  |
| start  process  stop  Display output  Read input | start    X=10,y=20  Z=x+y  Output z  stop | Begin the flowchart. |  |
| Initialize value for variables X& y. |  |
| Add X & Y and store the results in variable Z. |  |
| Output the value of the variables Z. |  |
| Terminate the flowchart. | |

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**The switch statement**

This is a **multiple-branching statement** where, based on a condition, the control is transferred to one of many possible points.

switch(expression)

{

casel:

{

actionl; break;

}

case2:

{

action2; break;

}

default:

{

action4; break;

}

}

action5;

|  |  |  |
| --- | --- | --- |
| **Selection structure** | **Example flowchart** | **Step by Step instructions** |
| True alternative  False alternative  Is condition true? | start  Output a  Is b>c  Is a>c  Isa>b  Reada,b,c  NO YES  YES  Output c  YES NO | Begin the flowchart |
| Read three no. and store A,B,C respectively |
| Check whether value of A is greater than the value of B |
| If A is greater than B, then check whether A is >then C, otherwise check whether B is>than C |
| Output c  NO  Output b |
| If A is > than C, displays A.otherwise displays C.however, if B is > than C,then display the value of B,otherwise display C |
| stop | | Connector symbols are used to join all the flow lines |
| Terminate the flowchart |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**The do-while statement**

The do-while is an **exit-controlled loop**. Based on a condition, the control is transferred back to a particular point in the program. The syntax is as follows:

do

{

actionl;

}

while(condition is true);

action2;

**The while statement**

This is also a loop structure, but is an entry-controlled one. The syntax is as follows:

while(condition is true)

{

actionl;

}

action2;

**The for statement**

The for is an **entry-enrolled loop** and is used when an action is to be repeated for a predetermined number of times. The syntax is as follows:

for(initial value; test; increment)

{

actionl;

}

action2;

|  |  |  |
| --- | --- | --- |
| Repetition structure | Example flowchart | Step by step instruction |
| Is conditiontrue?  No  Yes  Avg=sum/n | If count<N  print AVG  STOP  AVG=SUM/N  Sum=sum+A  count=count+1  Read A  Count=0, sum=0  Read n  start  YES  NO | Begin the flowchart. |
| Read 3 no. store value of A,B,C respectively. |
| Check whether value of A is greater than B. |
| if A is greater than B, then check A is greater than C, otherwise check whether B is greater than C . |
| IF A is greater than C, display A, otherwise display C. if B is greater than C, display B, otherwise display C. |
| connector symbols are join all the flow lines. |
| Terminate the flowchart. |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*