CHAPTER 2

OVERVIEW OF C++

2.1 Introduction

C++ was developed at AT & T Bell laboratories in the early 1980s by ***Bjarne Stroustrup.*** The name C++ (pronounced as C plus plus) was coined by Rick Mascitti where "++" is the C increment operator.

2.2 C++ character set

Like the C language, C++ also comprises a character set from which the tokens (basic types of elements essential for programming coding) are constructed. The character set comprises of "A" .. "Z" , "a" .. "z", 0 .. 9, +, -, /, *,\, (,), [,], {, }, =, !=, <, >, . , ' " ; : %, ! , &, ?, _, #, <=, >=, @, white space, horizontal tab, carriage return and other characters.

A quick recap of the basic types : The basic types are collectively called as **TOKENS**. A token is the smallest individual unit in a program. Tokens are classified as shown in Fig 2.1.

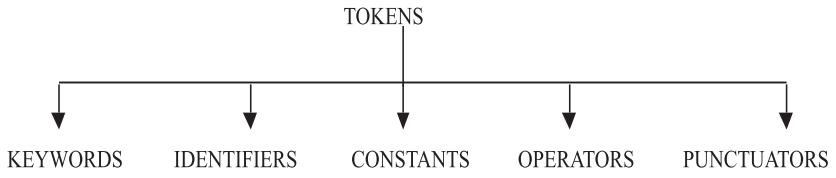


Fig. 2.1 Classification of Tokens

2.2.1 Keywords

Keywords have special meaning to the language compiler. These are reserved words for special purpose. These words cannot be used as normal identifiers. Table 2.1 shows the list of keywords used in C++.

auto	break	case	const	class	continue
default	delete	do	else	enum	for
friend	goto	if	inline	new	operator
private	protected	public	return	signed	sizeof
static	struct	switch	this	unsigned	virtual
while					

Table 2.1 Keywords

2.2.2 Identifiers

Identifiers are also called as variables. Variables are memory boxes that hold values or constants. A variable name must begin with an alphabet or underscore followed by alphabets or numbers. For example *_test ; test ; sum*12 are some valid identifiers. We shall see more about variables after dealing with data types

2.2.3 Constants

Constants are data items whose values cannot be changed. A constant is of numeric or non-numeric type. Numeric constants consist of only numbers, either whole numbers or decimal numbers. Integer, floating-point are numeric constants.

2.2.4 Integer Constant

- Integer Constant must have at least one digit and must not contain any fractional part.
- May be prefixed with a + or – sign
- A sequence of digits starting with **0** (zero) is treated as Octal constant Ex. 010 = 8 ($[8]_{10} = [10]_{8}$)
- A sequence of digits starting with **0x** is treated as hexadecimal integer. Ex. 0xF = 15 ($[15]_{10} = [F]_{16}$)

2.2.5 Floating Point Constant

Floating Point Constant is a signed real number. It includes an integer portion, a decimal point, a fractional portion and an exponent. While representing a floating point constant the integer portion or the decimal portion can be omitted but never both. For example 58.64 is a valid floating point (Real) constant. It can be represented in exponent form as follows :

- 5.864E1 => 5.864 x 10^{1} => 58.64
- 5864E-2 => 5864 x 10^{-2} => 58.64
- 0.5864E2 => 0.5864x10^{2} => 58.64

The letter E or e is used to represent the floating-point constant exponent form.

2.2.6 Character Constant

Character constant is a constant that contains a single character enclosed within single quotes. It can be any character as defined in the character set of C++ language (alphabet, numeral, mathematical, relational or any other special character as part of the ASCII set). Certain special characters like tab, backspace, line feed, null, backslash are called as non-graphic character constants. These characters are represented using escape sequences. Escape sequences are represented using characters prefixed with a backslash. Table 2.2 shows the escape sequences.

Escape Sequence	Nongraphic Character
\a	Bell
\b	Back space
\n	New line/ line feed
\t	Horizontal tab
\v	Vertical tab
\\	Back slash
\' or \"	Single / double quotes
\o	Octal number
\x	Hexadecimal number
\0	Null

Table 2.2 Escape Sequences

2.2.7 String Literal

String Literal is a sequence of characters surrounded by double quotes. String literals are treated as array of characters. Each string literal is by default added with a special character '\0' which marks the end of a string. For example "testing"

2.2.8 Operator

Operator specifies an operation to be performed that yields a value. An operand is an entity on which an operator acts. For example :

RESULT = NUM1 + NUM2

NUM1 and NUM2 are operands. **+** is the additional operator, that performs the addition of the numbers. The result (value) generated is stored in the variable RESULT by virtue of "=" (Assignment) operator. Table 2.3 shows the operators in C++.

[ ]	*	%	==	=	>=
( )	+	<<	!=	*=	&=
.	-	>>	^	/=	^=
->	~	<	|	+=	|=
++	!	>	&&	-=	, --
	size of	<=	||	%=	#
&	/	>=	?:	<<=	##

Table 2.3 Operators in C++

The following operators are specific to C++.

:: .* ->*

The operators # and ## are used only by the preprocessor.

Operators are classified as

- Arithmetic
- Assignment
- Component Selection
- Conditional
- Logical
- Manipulator
- Member dereferencing
- Memory Management
- Preprocessor
- Relational
- Scope Resolution
- Shift
- Type Cast

Based on operand requirements, operators are also classified as unary, binary and ternary operators.

For example :

Unary operators require one operand
Binary operator requires two operands
Ternary operator requires three operands.

Table 2.4a
Unary Operators

&	Address of
!	Logical Not
*	Indirection
++	Increment
~	Bitwise
--	Decrement
-	Unary minus
+	Unary plus

Table 2.4b Binary Operators

Additive	+	Binary Plus
	-	Binary minus
Multiplicative	*	Multiply
	/	Divide
	%	Remainder (Modulus)
Shift	<<	Shift Left
	>>	Shift Right
Bitwise	&	AND
	|	OR
	^	XOR
Logical	&&	Logical AND
	||	Logical OR
Assignment	=	Assignment
	/=	Assign quotient
	+=	Assign sum
	*=	Assign product
	%=	Assign remainder
	-=	Assign difference
	<<=	Assign left shift
	>>=	Assign right shift
	&=	Assign bitwise AND
	^=	Assign bitwise XOR
	|=	Assign bitwise OR
Relational	<	Less Than
	>	Greater than
	<=	Less than or Equal to
	>=	Greater that or Equal to
Equality	==	Equal to
	!=	Not Equal to
Component Selection	.	Direct Component Selection
	->	Indirect Component Selection
Class member	::	Scope access/Resolution operator
	.*	Dereference Operator
	->*	Dereference pointer to class member
Conditional	? :	Ternary operator
Comma	,	Evaluate

2. 2.7.1 Arithmetic Operators

Arithmetic Operators are used to perform mathematical operations. The list of arithmetic operators are :

- +
- -
- * multiplication operator
- / division operator
- % modulus operator - gives the remainder of an integer divison
- += , -=, *= , /= , %=

Arithmetic expressions are formed using arithmetic operators, numerical constants/variables, function call connected by arithmetic operators.

Examples :

- a = -5;
- a = +b;
- a /= 5; (a = a/5)
- a++; (Post increment operator . Equivalent to a = a+1)
- a— ; (Post decrement operator. Equivalent to a = a-1)
- ++a; (Pre increment operator. Equivalent to a = a+1)
- —a ; (Pre decrement operator. Equivalent to a = a – 1)
- a *= 2 (a = a * 2)
- a %= 5 (a = a/5)
- a = b + pow(x,y) (a = b + x^y)

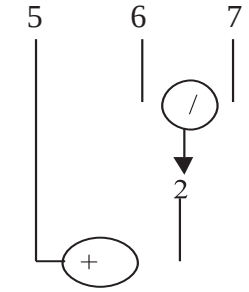
Operators are executed in the order of precedence. The operands and the operators are grouped in a specific logical way for evaluation. This logical grouping is called as association. Table 2.5 indicates the Mathematical Operators, its Type, and Association.

Operator Precedence	Type	Associativity
() []	Mathematical	Left to right
Postfix ++, -- ,	- Unary	Left to right
prefix ++, --		Right to left
+unary , - unary	mathematical	Right to left
		Right to left
* / %	Mathematical –binary	Left to right
+ -	Mathematical– binary	Left to right

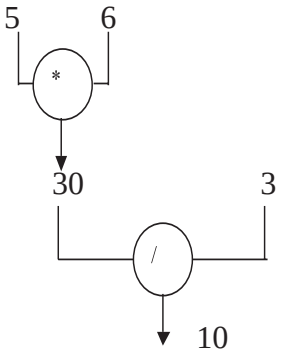
Table 2.5 Mathematical Operator Precedence

The following examples demonstrate the order of evaluation in arithmetic expressions :

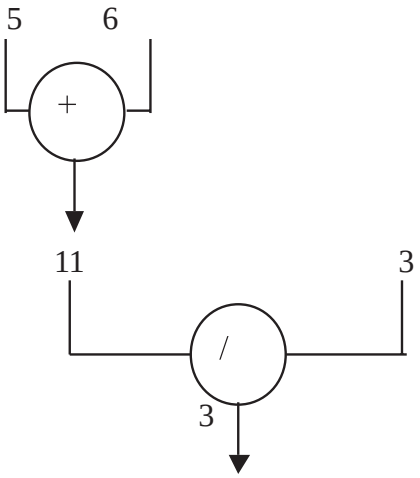
5 + 6/3 will yield the result as 7



5 * 6 / 3



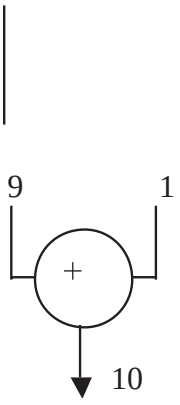
(5 + 6) / 3

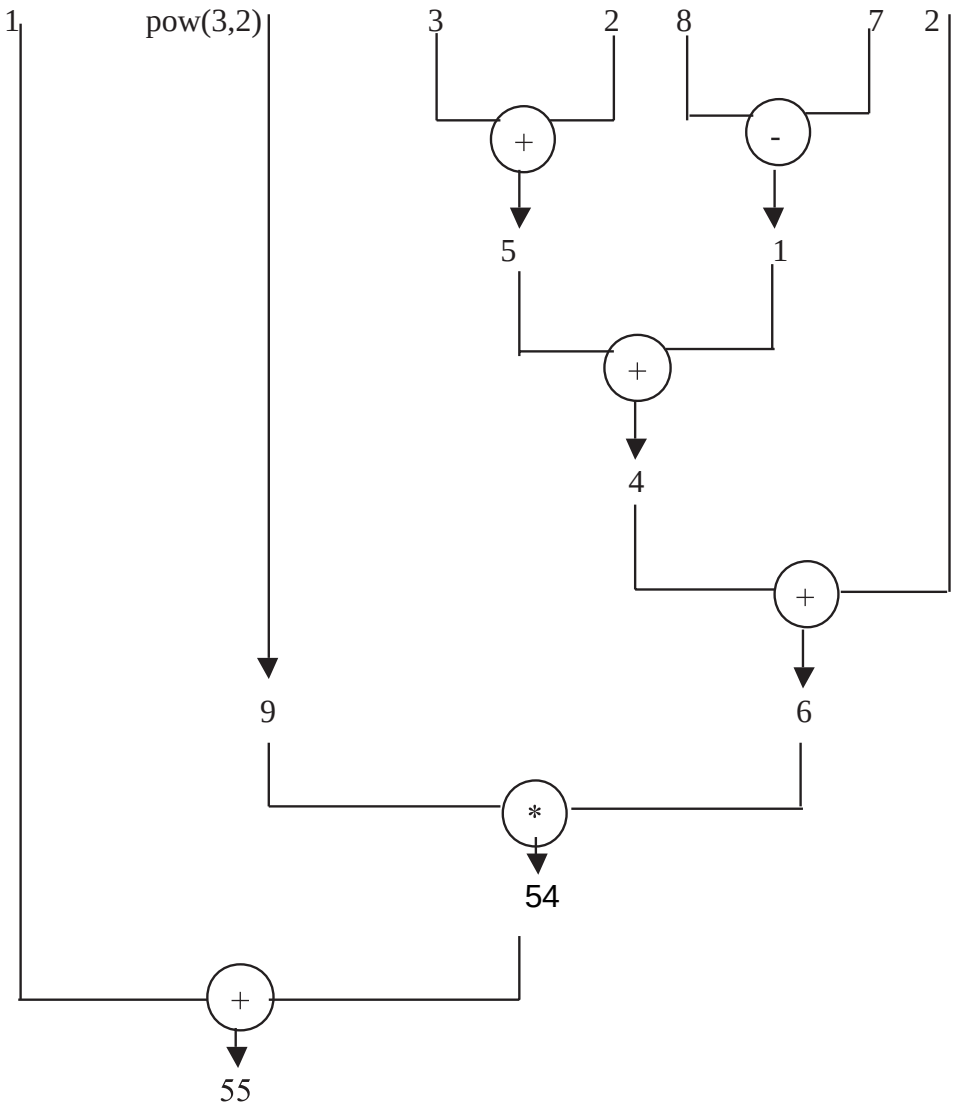


The result is 3, as all the inputs are of integer type.
The result will be 3.66 if any of the inputs are of float type.

1 + pow (3 , 2)

pow(3 , 2)





Increment and Decrement operators are unique to C++. Evaluation of expressions using these operators are indicated in the Table 2.6 .

Expression	Operation	Example

a++	Get the value of a, then increment the value of the variable by 1	a =5;
		c = a++
		Execution :
		c = a;
		a = a+ 1;
		Hence the value stored in the variable c is 5.

++a	Increment the value of the variable a by 1, and then get the value	a = 5;
		c = ++a;
		Execution:
		a=a+1
		c = a;
		Hence the value of c will be 6

a--	Get the value of a , then decrement it by 1	a = 5;
		c = a--;
		Execution :
		c = a;
		a = a –1
		What will be the value of c ?

--a	Decrement the value of a by 1, then get the value of a	a = 5
		c= --a;
		Execution :
		a = a – 1
		c = a;
		What will be the value of c ?

Table 2.6 Increment and Decrement Operator

What will be the values stored in the variables of the following snippets as shown in Table 2.7?

1. a = 5 b = 5 a = a + b++ Value stored in the variable a is _____	2. x = 10 f = 20 c = x++ + ++f Value stored in the variable c is _____ x is _____ f is _____	3. fun = 1 sim = 2; final = —fun + ++sim – fun - Value stored in the variable fun is _____ sim is _____ final is _____

Table 2.7 Simple Problems

2.2.7.2 Relational Operators

Relational Operators are used to compare values. The list of relational operators are :

- = = equal to
- > greater than
- < lesser than
- >=, <= greater than or equal to , lesser that or equal to
- != not equal to

Relational operators are used to compare numeric values. A relational expression is constructed using any two operands connected by a relational operator. For example the relational operators are used to construct conditions such as

- 10 > 20
- 500.45 <= 1005
- 99 != 99.5
- 9 = = 9

The result of a relational operation is returned as true or false. The numeric constant zero (0) represents False value, and any non-zero constant represents true value. The above expressions output will be

- 0 (10 > 20 is false) ;
- 1 (500.45 < 1005 is evaluated to True hence any non zero constant) ;
- 1 (99 != 99.5 will be evaluated to True hence non zero constant)
- 1 (9 = = 9 will be evaluated to true, hence the output will be non zero constant)

What will be the value of the following expression ?

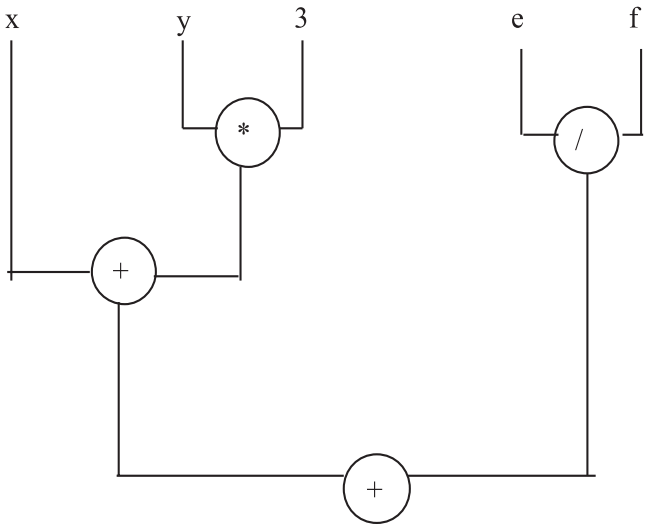
```
( num1 + num2 – num3 )/5 * 2 < ( num1 % 10)
where num1 = 99 , num2 = 20, num3 = 10
```

Evaluate the relational expressions shown in the following Table 2.8.

Operator	Expression	Result
= =	5 = = 6	0
! =	‘a’ = = ‘a’	
>	5 > 6	
	‘a’ > ‘A’	
<	5 < 6	
	‘a’ < ‘A’	
>=	‘a’ >= ‘z’	
	5 >= 5	
<=	‘a’ <= ‘z’	
	5 <= 6	

Table 2.8 Evaluate Relational Expressions

Relational operators have lower precedence than the arithmetic operators. For example the expression **x + y * z < e / f** will be evaluated as follows :



2.2.7.3.Logical Operators (Boolean Operators)

Logical operators combines the results of one or more conditions. The various logical operators are && (AND) , || (OR) , ! (NOT)

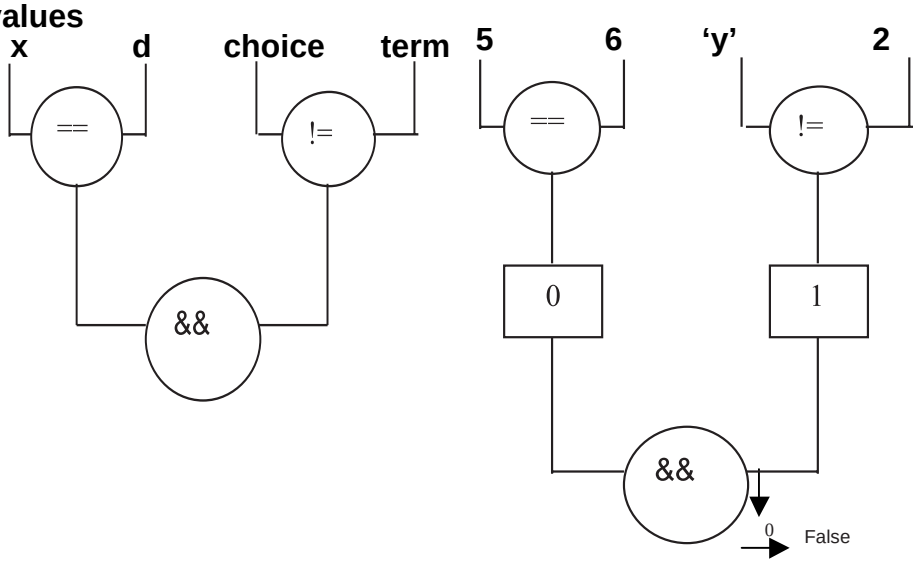
Example : c = 5 , d = 6 , choice = 'y', term = '2' (Assume True is indicated as 1 and False as 0)

Result_1 = (c = = d) && (choice != term)
Result_2 = ('y' = = 'Y') || (term != '0')

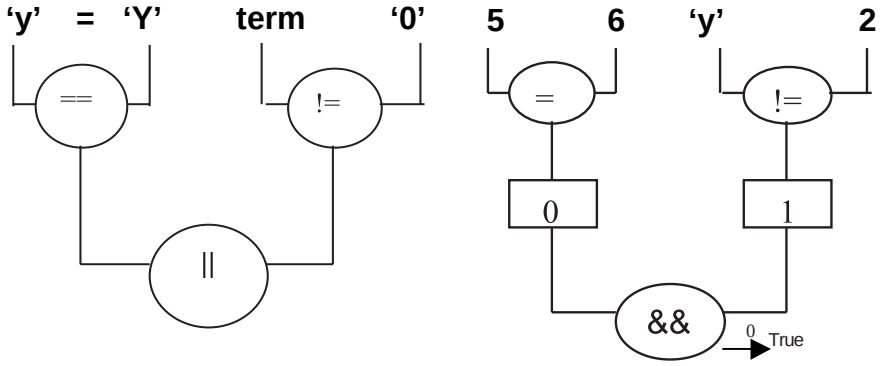
Result_3 = (c = = d) && '(y' = = 'Y')|| choice != term
Result_4 = (c = = d) || ('y' = = 'Y') && choice ! = term

What will be the values stored in Result_1 and Result _2 ??
The values stored in Result_1 is 0 (False) ; Result_2 is 1 (True) , Result_3 is 1 (True) and Result_4 is 0 (False).

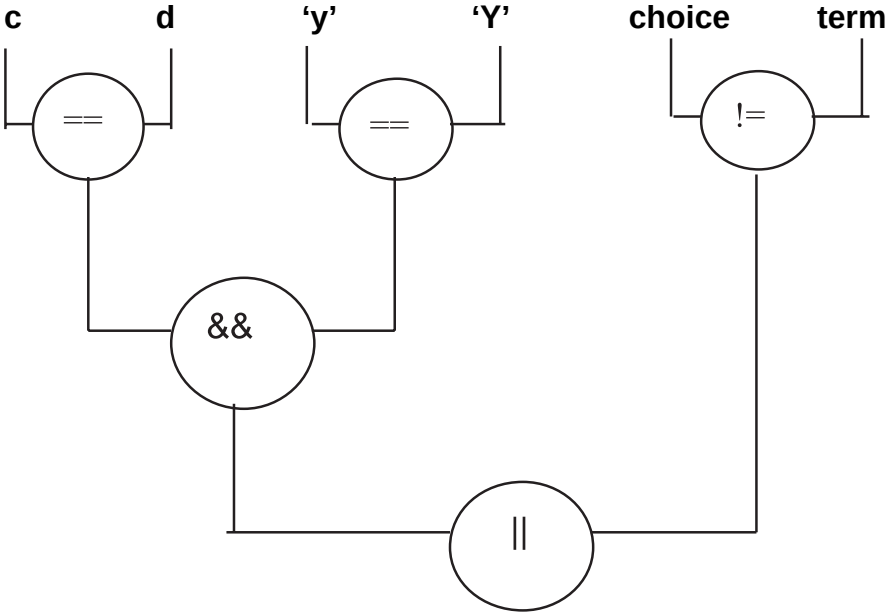
Result_1= (c==d) && (choice != term) By substituting values



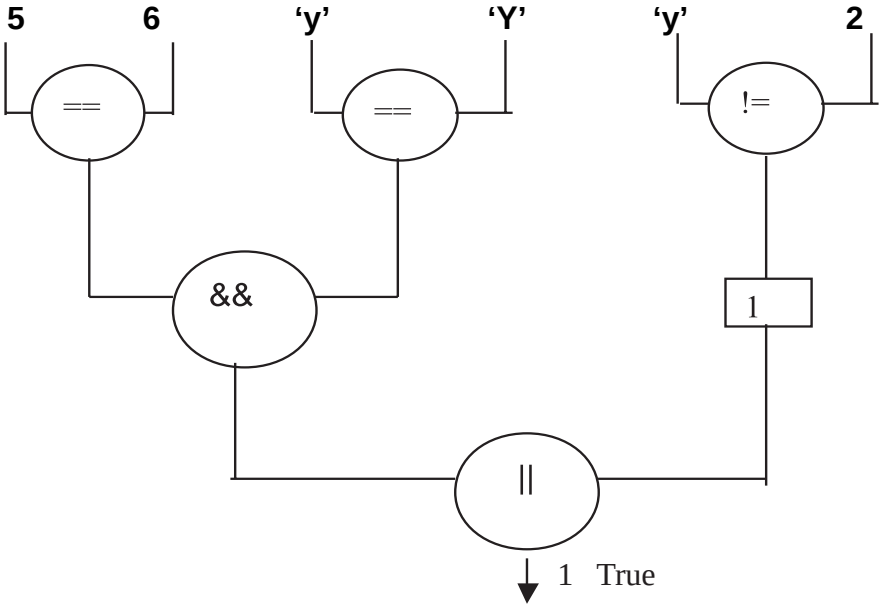
Result_2 = ('y'=='Y') && (term != 0) By substituting values



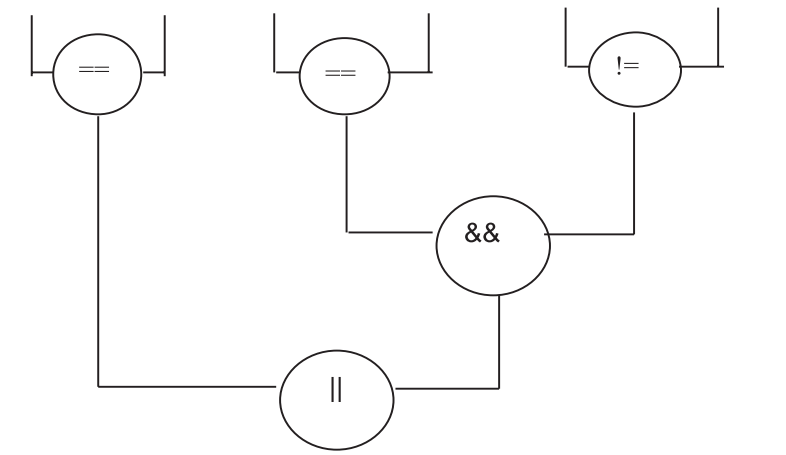
Result_3 = (c==d) && (‘y’ == ‘Y’) || (choice != term)



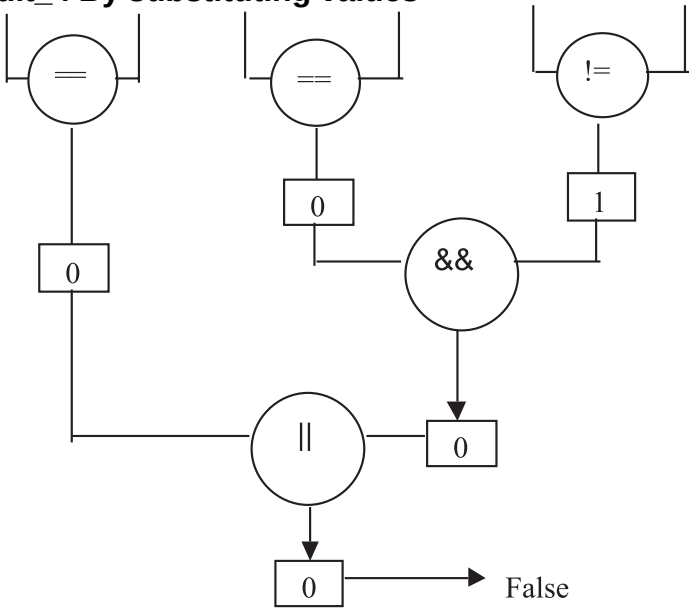
Result_3 By substituting values



Result_4 = (c == d) || (‘y’ == ‘Y’) && choice != term

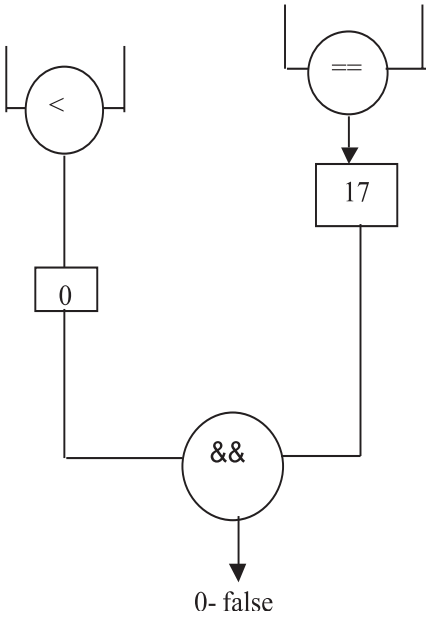


Result_4 By substituting values



The Logical operators have lower precedence to relational and arithmetic operators. Can you evaluate the value of the following expression ?

5 < 4 && 8 + 9



2.2.7.4. Conditional Operator (?:)

(num1 > num2) ? “true”:”else” - ?: Is a ternary operator – (num1>num2,”true”,”false” are the operands. A ternary operator (?:) is also called as conditional operator. The general syntax is E1 ? E2 : E3 where E1,E2,E3 are operands. E1 should essentially be of scalar type, E2 and E3 are values or statements. For example to assign the maximum value of the two values one can express it as :

max = (num1 > num2) ? num1 : num2; The variable **max** will take the value of num1 if num1 is greater than num2, otherwise max will be assigned with the value of num2.

Can you write out what will be the value stored in **x** of the following snippet ?

```
a = 10
b = 10
x = ( a < b) ? a*a : b % a;
```

2.2.7.5. Assignment Operators

= is the simple assignment operator. It is used to assign the result of an expression (on the right hand side) to the variable (on the left hand side of the operator). In addition to the simple assignment operator, there are 10 'shorthand' assignment operators . Refer to the Table 2.9 for all assignment operators.

Expression	**Working**	**Result**
A = 5	The value 5 is assigned to the variable A.	The variable takes the value 5.
A += 2	A+= 2 is interpreted as A = A +2	The value stored in Ais 7
A *= 4	A = A * 4	The value stored in A is 20
A /= 2	A = A / 2	The value stored in A is 2
A - = 2	A = A – 2	The value stored in A is 3
A %= 2	A = A % 2	The value stored in A is 1
Evaluate the following expressions where a = 5, b = 6, c = 7		
A += b*c C *= a + a / b B += a % 2 * c		

Table 2.9 Assignment Operators

Table 2.10 gives the complete Operator precedence of all operators used in C++

Operator Precedence	Type	Associativity
() []		
Postfix ++, — , prefix ++, — ! –logical not + unary , - unary	Mathematical-Unary Logical – unarymathematical	Left to right Left to right Right to left Right to left Right to left Left to right
* / %	Mathematical –binary	Left to right
+ -	Mathematical– binary	Left to right
< <= > >=	Relational-binary	Left to right
= = !=	Relational-binary	Left to right
&& (AND)	Logical – binary	Left to right
|| (OR)	Logical – binary	Left to right
?:	Logical – ternary	Left to right
= *= /= %= += -= <<= >>= &= ^= |=	Assignment	Right to left

Table 2.10 Operator Precedence

(note : operators specific to C++ will be dealt with in their relevant topics)

2.2.8 Punctuators

Punctuators are characters with a specific function. Refer to the Table 2.11 for Punctuators and their Purpose.

Punctuators	Purpose
;	Terminates a C++ statement
//	Treats statements prefixed with this as comments
/* */	Blocks enclosed within these characters are treated as comment
{ }	Used to group a set of c++ statements. Coding for a function is also enclosed within these symbols
[ ]	Index value for an element in an array is indicated within these brackets
‘ ’	Is used to enclose a single character
“ “	Is used to enclose a set of characters

Table 2.11 Punctuators and their Purpose

2.3 Data Types

Data Types are the kind of data that variables hold in a programming language. The ability to divide data into different types in C++ enables one to work with complex objects. Data is grouped into different categories for the following two reasons :

- The compiler may use the proper internal representation for each data type

- The programmer designing the programs may use appropriate operators for each data type. They can be broadly classified into the following three categories.
  - User defined type
  - Built-in type
  - Derived type

The broader classification is indicated in the Fig. 2.2

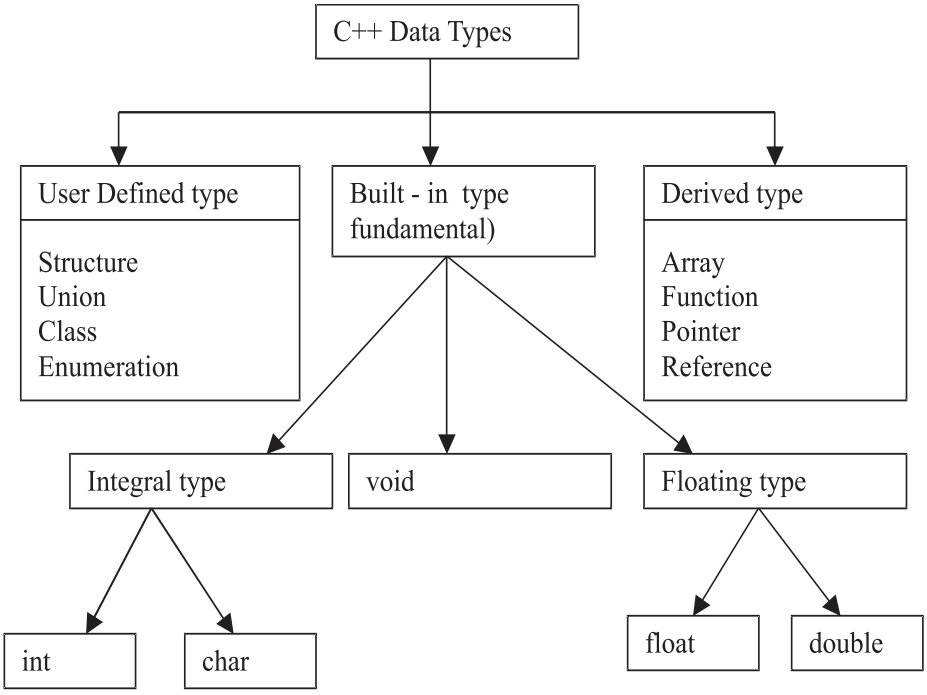


Fig. 2.2 C++ Data Types

2.3.1 User Defined Data Type

User Defined Data Type enables a programmer to invent his/ her own data type and define values it can assume. This helps in improving readability of the program.

For example consider the following user defined data type

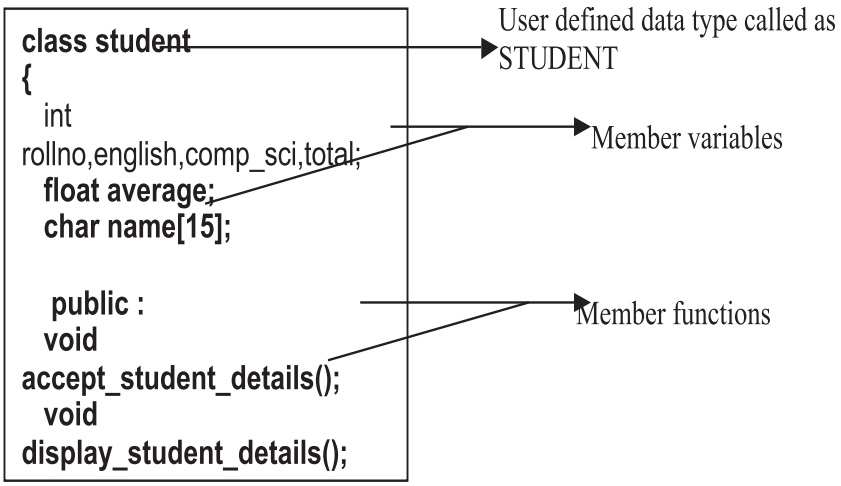


Fig. 2.3 User Defined Data Type

student is a user defined data type of **class**. This data type defines the features of a student in terms of member variables, and the associated functions like accepting data for a student, displaying details, and also calculating their respective totals and averages. Thus the **class student improves the credibility and readability of the program by combining** the data requirements and its associated functions in the form of a data type for a student.

Users can define a variable that would represent an existing data type. "**Type definition**" allow users to define such user defined data type identifier. The syntax :

typedef data_type user_defined_data_type_identifier;

For example:

typedef int **marks**;
typedef char **grade**;

The data type identifiers **marks** and **grade** are user defined identifiers for int and char respectively. Users can define variables of int and char as follows:

marks eng_marks, math_marks;
grade eng_grade, math_grade ;

typedef helps in creating meaningful data type identifiers, that would increase the readability of the program.

Another user defined data type is they enumerated data type. As the name suggests, enumerated data type helps users in creating a list of identifiers, also called as symbolic numeric constants of the type int.

The syntax :
enum data type identifier (value 1, value 2, … value n);

Examples :
enum **working_days** (Monday, Tuesday, Wednesday, Thursday, Friday);
enum **holidays** (Sunday, Saturday);

The identifiers **working_days** , **holidays** are user defined data type. Monday, Tuesday … is the list of values also called as **enumeration constants** or **numeric constants.**

Users can declare variables of this enumerated data type using the syntax :

enum identifier variable1, variable2 …,variable n;

For example the variables first_workingday and last_workingday of the type working_days may be declared as follows:

working_days **first_workingday**, **last_workingday**;

These variables can take only one of the values defined for working_days.

```
first_workingday = Monday ;
last_workingday = Friday;
```

The enumeration constants (Monday, Tuesday, Wednesday…) are given integer constants starting with 0 (zero) by the compiler. The above assignment statements can also be rewritten as:

```
first_workingday = 0 ;
last_workingday = 4;
```

Users can also redefine these integer constants by assigning explicit values to the enumeration constants as

enum **working_days** (Monday = 1, Tuesday, Wednesday, Thursday, Friday);

Here, the constant Monday is assigned the value of 1. The remaining constants are assigned successive integer constants.

2.3.2. Storage Class

Storage Class is another qualifier (like long or unsigned) that can be added to a variable declaration. The four storage specifiers are **auto**, **static**, **extern** and **register**. static and register variables are automatically intialized to zero when they are declared. Auto variables are not initailized with appropriate values based on their data type. These variables get undefined values known as garbage. The following Table 2-12 gives the meaning and relevant examples.

Storage	Meaning	Example

auto	Defines local variable known to the block in which they are defined. By default the local variables are auto hence rarely used.	void main()
		{
		autofloat ratio;
		int kount;
		}
		
		The variables ratio and kount defined within the function main() have the storage specifier as auto.

static	Variables defined within a function or a block cease to exist , the moment the function or the block looses its scope. **Static** modifier allows the variable to exist in the memory of the computer, even if its function or block within which it is declared looses its scope. Hence the variable also retains the last assigned value.	void fun(){
		static int x;
		x++;
		}

extern	Global variable known to all functions in the current program. These variables are defined in another program.	extern int filemode;extern void factorial();
register	The modifier register instructs the compiler to store the variable in the CPU register to optimize access.	void fun(){ register int I;}

Table 2.12 Storage Classes

2.3.4 Built in Data Types

Built in Data Types are also called as Fundamental or Basic data types. They are predefined in the compiler. Integral, Float and Void are the three fundamental data types.

Integral type is further divided into **int** and **char.** Int is the Integer data type. It cannot hold fractional values. **char** is character data type that can hold both the character data and the integer data. For example consider the declaration and initialization of the variable **ch -** char ch = 'A'. The statement char ch = 65 would also yield the same result of storing the value 'A' in the variable ch as character data type can hold both character and integer values.

Floating type is further divided into **float** and **double.** Floating type can store values with fractional part (Refer to floating point constants representation)

Void type has two important purposes :

- To indicate the a function does not return a value
- To declare a generic pointer

For example consider the following functions defined in C++ (Program void.cpp & fun.cpp).

Program void.cpp

```
#include<iostream.h>
#include<conio.h>
void  fun(void)
{
  int a,b;
  cin >> a >> b;
  cout << a+b;
}

void main()
{
  fun();
}
```

Program fun.cpp

```
#include<iostream.h>
#include<conio.h>
int fun(int a, int b)
{
  return  a+b;
}
void main()
{
   int a = 5 , b = 6, sum = 0;
   sum =  fun(a,b);
   cout << sum;
}
```

In the example void.cpp the prototype void fun(void) indicates that the function does not return any value, nor does it receives values(in the form of parameters). Hence the call statement in the main() function is given as **'fun()'** . In the example fun.cpp, the prototype **int fun(int a, int b)** , instructs the compiler that the function fun() returns an integer value. Hence the call statement in the main() function is given as '**sum = fun(a,b)'** The variable sum receives the value from the return statement (return a+b)

- ✓ **void** data type indicates the compiler that the function does not return a value, or in a larger context void indicates that it holds nothing.

Basic data types have several modifiers. These modifiers have a profound effect in the internal representation of data. signed, unsigned, long and short are some of the modifiers. Table 2.13 gives a list of the data types, memory allocation and range of values.

2.3.4. Derived Data Type

These are built from the basic integer and floating type (built in type) or user defined data types. For example

```
int num_array[5]      =   {1,2,3,4,5};
chardayname[7][3]     =  {Sun","Mon","Tue","Wed","Thu",
                          "Fri","Sat"};
```

num_array stores 5 values. Each element is accessed using the positional value of the element in the array. The position numbering commences from zero. num_array[0] stores value 1 and num_array[4] stores value 5.

Can you write as to what is stored in dayname[0],dayname[5] and dayname[3][2] ?

Type	Byte	Range
char 1	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2	-32768 to 32767
unsigned int, unsigned short int	2	0 to 65535
signed int,short int, signed short int	2	-32768 to 32767
long int,signed long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float 4	4	3.4e-38 to 3.4e+38
double	8	1.7e-308 to 1.7e+308
long double	10	3.4e-4932 to 1.1e+4932

Table 2.13 Data Types Size & Range of Values

2.3.4.1 Pointers

A pointer is a variable that holds a memory address. Pointers provide the means through which the memory locations of a variable can be directly accessed. Every byte in the computer's memory has an address. Addresses are numbers just as our house numbers. The address number starts at NULL and goes up from there.. 1, 2 , 3…..

For example a memory size of 640 KB will have addresses commencing from NULL and goes up to 655, 358 as shown in Fig. 2.4.

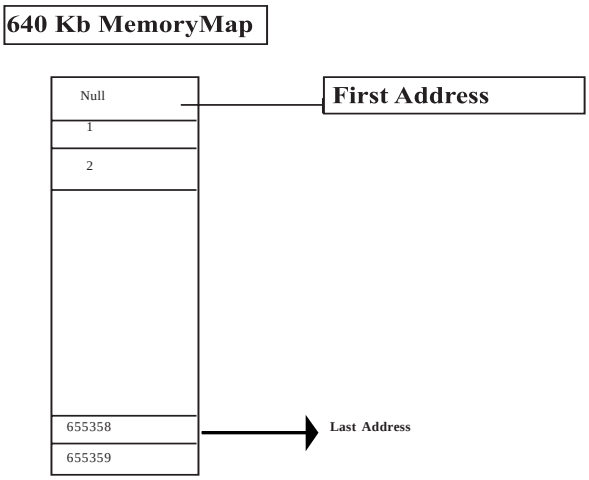
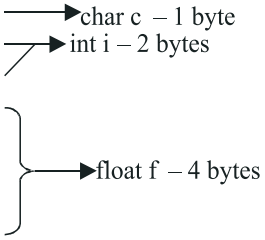


Fig. 2.4 640 Kb Memory Map

When a program is compiled, some memory is allocated to the variables by the compiler. The amount of memory allocated to each variable depends on the data type of the variable.

For example consider the declarations :
char c ; int i; float f;



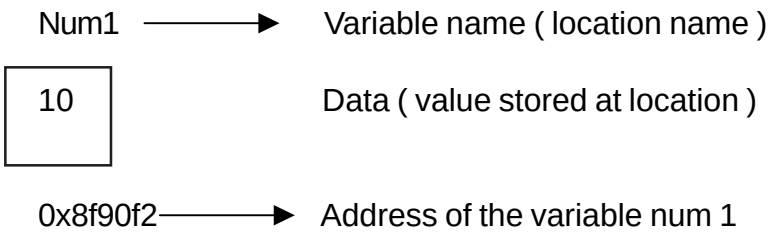
Every variable will be referred by its address. From our example the address of the variables c,i and f will be 1,2 and 4 respectively. Addressing is done using the hexadecimal system.

When dealing with pointer data type one needs to know about the **address of (&)** operator and the **value at operator (*).**

The ‘ &’ operator : When we type int num1=10;

the C++ compiler performs the following operations / actions :zz

1) Reserves space in the memory to hold the integer value
2) Associates the variable name num1 with a memory location
3) Stores the value 2 at this location in the memory




```
// Program – 2.1
// to demonstrate use of & operator
#include<iostream.h>
#include<conio.h>
void main()
{     clrscr();
      int i = 10;

cout << "\n Address of the variable... " <<&i;
cout << "\nValue stored in the variable .." << i;

      getch();
}
```

Now consider this

```
int *x , num1;
num1=10;
x=&num1;
cout<<*x;
```

Note :

The asterix (*) is

1) Used to declare a pointer variable
2) Used to display the contents stored at a location (value at the address operator)
3) It is a unary operator

2.4 Variables

The name assigned to a data field that can assume any of a given set of values is defined as the variable. For example consider the following group of statements

```
int num;
num = 5;
```

The statement **int num**; may be interpreted as " num is a variable of the type integer ". The assignment statement **num = 5** may be interpreted as the value 5 is stored in the variable num.

Variables are user defined named entities of memory locations that can store data.

Variable names may contain letters, numbers and the underscore character(_). Names must begin with a letter or underscore. (However names beginning with an underscore are reserved for internal system variables). Names are case sensitive, which means that it differentiates between lower case and upper case letters.

Complete the Table – 2.14.

Variable	**Valid/Invalid**	**Reasons if invalid**
A_b	Valid	
1a_b	Invalid	Variables must begin with an alphabet or an underscore only.
_Test		
Balance$		
#One		
Include		

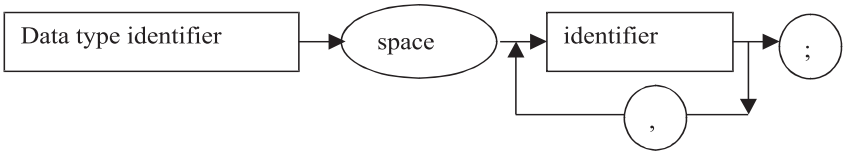
Table 2.14 Validity of Variable Names

2.4.1 Declaration of Variables

Variables are allocated memory to store data. Compiler allocates memory, based on the data type of the variable. Hence variables must be declared before they are used.

Example : int a;
float f1,f2;
char name[10],choice;

Syntax :



Consider the declaration **int side, float hypotenuse , area ;** This is an erroneous declaration because the compiler interprets this statement as follows :

- The variables side, float, hypotenuse and area will be treated as instances of the data type **int**. Hence it throws an error stating that " **comma is expected after float**"
- The intention was to declare the variable side of int data type and the vairbales hypotenuse & area of the data type float.
- Hence the above declaration statement should be rewritten as follows :

```
int side ;
float hypotenuse , area ;
```

```
int side ; float hypotenuse,area ;
```

✓ **More than one variable of the same data type can be declared in a single declaration statement. But every variable should be separated by comma.**

There are nine words for data types such as **char , int , double , float, void, short, signed, long and unsigned .**

long, short, signed and unsigned are qualifiers or modifiers that modify a built – in data type with the exception of void.

The internal representation for the integer value 15 is **0**0000000 00001111 . Integer values are stored in 16 bit format in binary form. Starting from right extreme, 15 bits are used to store data. Maximum value stored in an integer variable is +32767 and the minimum value is –32768, as 2^{15} **is 32767 on the positive side and –32768 on the negative side. 16th bit, also called as the Most Signigicant Bit or sign bit. It is used to store sign. The 16th bit will have a value 1 if**

negative value is stored. When the modifier unsigned is used the integer data type will store only positive values, the sign bit is also used to store data. Therefore the range to store data goes upto 2^{16} , hence the maximum value will be 65535.

> ✓ **The modifier alters the base data type to yield new data type.**

The impact of modifiers :

- **unsigned** modifies the range of the integer values as the sign bit is also used to store data.
- **long** increases the bytes for a particular data type, thus increasing the range of values.

The base data type should be prefixed with the modifiers at the time of declaring a variable. For example :

unsigned int registration_number;
long unsigned int index;
short signed char c;

> ✓ **Prefix the data type with modifiers at the time of declaring variables.**

The **const** qualifier specifies that the value of a variable will not change during the run time of a program. Any attempt to alter the value of a variable defined with this qualifier will throw an error message by the compiler. The const qualifier is used like any other modifier where the variable is prefixed with the keyword const followed by data type .

For example :

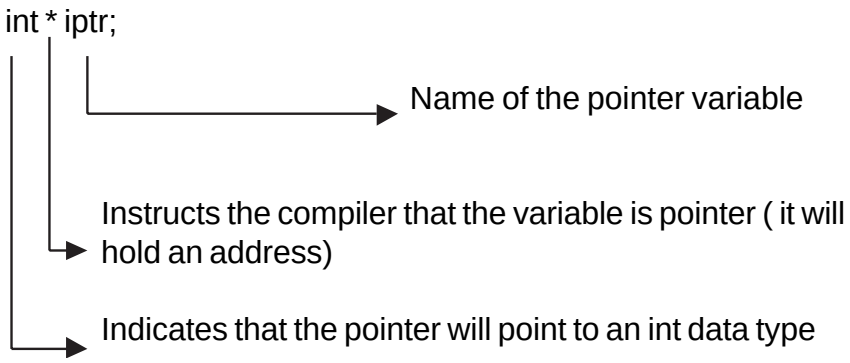
const float pi = 3.14;

The Table 2.15 shows the **data types** with altered lengths and range of values when the qualifiers or modifiers are used

Data Types		
Type	Length	Range
unsigned char	8 bits	0 to 255
char	8 bits	-128 to 127
enum	16 bits	-32,768 to 32,767
unsigned int	16 bits	0 to 65,535
short int	16 bits	-32,768 to 32,767
int	16 bits	-32,768 to 32,767
unsigned long	32 bits	0 to 4,294,967,295
long	32 bits	-2,147,483,648 to 2,147,483,647
float	32 bits	3.4 *(10**-38)to 3.4 * (10**+38)
double	64 bits	1.7 *(10**-308)to 1.7*(10**+308)
long double	**80 bits**	**3.4 * (10**-4932) to 1.1 * (10**+4932)**

Table 2.15 Data Types with Modifiers

Declaring pointer variables



The declaration statement int *ptr may be read as ptr is a pointer variable of the type int. The variable ptr can only store addresses that hold integer values.

Examples of pointer variable declarations:

char * cptr float * fptr	declaring a pointer to character type a pointer to float type
void *v ptr	a pointer that can point to any data type a generic pointer is declared in this way
const int * ptr	ptr is a pointer to a constant integer (cannot modify the value stored at the address pointed by ptr)
char * const cp	cp is a constant pointer. The address stored in cp cannot be modified

Table 2.16 Examples of Pointer Variables

2.4.2 Initialization of variables

Variables are initialized to a specific value at the time of declaration. Initialization is done only once. For example :

```
int  num = 10;
int fun(5)
```

In statement (1) the variable num is initialized to 10, whereas in the second statement the variable fun is initialized to 5 through a constructor.

Implicit conversions: refers to data type changes brought about in expressions by the compiler. For example consider the following snippet:

```
float f = 7.6;
int  x = f;
```

The value stored in the variable x is 7, as float is converted to int. The compiler does this conversion automatically.

Rules for implicit conversion :

Consider a term, having a pair of operands and an operator. The conversions takes place as follows :

1. If one operand is of type **long double** , then the other value is also converted to long double.
2. If one operand is of type **double,** then the other value is also converted to double.
3. If one of the operands is a **float**, the other is converted to a float.
4. If one of the operands is an **unsigned long int**, the other is converted to unsigned long int.
5. If one of the operands is a **long int**, then the other is converted to long int.
6. If one of the operands is an **unsigned int**, then the other is converted to an unsigned int.

```
// demonstrating implicit type conversions
        // Program - 2.2
# include <iostream.h>
# include <conio.h>
# include <iomanip.h>

void main()
{
 clrscr();

 int i;
 float f;
 double d;
 long double ld;
```

```
 unsigned int ui;
 unsigned long int uli;
 i = -5;
 f = 2;
 d = 3;
 ld = 3;
 ui = 6;
 uli = 4;
cout <<"\nSizeof long double.."<<sizeof(ld*d)<<'\t'<<ld*d;
cout << "\nSizeof double..." << sizeof(d*f)<<'\t'<<d*f;
cout << "\nSizeof float..." << sizeof(f * i)<<'\t' << f*i;
cout << "\nSizeof unsigned long int ..."
     << sizeof(uli* f)<<'\t'<< uli * f;
cout << "\nSizeof unsigned int..." << sizeof(ui * i)
     <<'\t'<< ui * i;
getch();
}
```

Note : **sizeof** is an operator . It returns the size (memory requirement) in terms of bytes, of the given expression or data type.

```
Output displayed by the above program:
Sizeof long double       ...10  9
Sizeof double            ...8   6
Sizeof float             ...4   -10
Sizeof unsigned long int ...4   8
Sizeof unsigned int      ...2   65506
```

Complete the following Table 2.17 based on the sample program 2.2 , write answers as shown in the reason column for the first value.

Sno.	Size of the result - in terms of bytes	Expression	Reason
1.	10	ld*d	The value generated is of long double type as the variable ld is long double. As long double data type requires 10 bytes to store a value, 10 is displayed.
2.	8	d*f	The value generated is of double type. …….
3.	4	f*l	
4.	4	uli * f	
5.	2	ui * i	

Table 2.17 Exercise based on Program 2.2

Initialization of pointer variables

Pointer variables can store the address of other variables. But the addresses stored in pointer variables are not of the same data type as this pointer variable is pointing to. For example :

```
int *iptr, num1;
num1 = 10;
iptr = &num1; // initializing a pointer variable
```

The following initalization is invalid.

```
int *iptr;
float num1 = 10.5;
iptr = &num1 // initializing  pointer variable pointing to integer
                data type with the address of float variable would
                 throw an error.
```

Pointer variables are sensitive to the data type they point to.

Type cast : Type cast refers to the process of changing the data type of the value stored in a variable . The statement **(float) 7** , converts the numeric constant 7 to float type. Type cast is achieved by prefixing the variable or value with the required data type. The syntax is : (data type) <varaible/value> or data type (variable/constant) . Type cast is restricted only to fundamental or standard data types. The statement **x = 8 % 7.7** will throw an error on compilation as, modulus operator % operates on integer data type only. This erroneous statement can be corrected as x = 8 % **(int) 7.7** - the float constant 7.7 is converted to integer constant by type casting it.

Complete the following Table 2.18

int x; x =7 / 3;	What is the value stored in X?
float x; x = 7 / 3;	What is the value stored in X?
float x; x = 7.0 / 3.0;	What is the value stored in X?
float x; x = (float) 7 / 3;	What is the value stored in X?

float x;
int a = 7 , b = 3;
x = a/b;
	What is the value stored in X?

float x;
int a = 7 , b = 3;
x = a/ (float) b;
	What is the value stored in X?

Table 2.18 Find the value of X

Exercises

1. Determine the order of evaluation of the following expressions :

   i. a + pow(b,c) * 2
   ii. a || b && c
   iii. a<b && c || d > a
   iv. (c>=50)||(!flag)&&(b+5 == 70)
   v. (a+b)/(a-b)
   vi. (b*b) – 4 * a * c

2. Identify errors in the following programs ..

a.
```
# include <iostream.h>
void main()
{
 float f = 10.0;
 x = 50;
 cout << x << f;
}
```

b.
```
# include <iostream.h>
void main()
{
 float f = 10.0;
 x = 50;
 cout << x << f;
}
```

c.
```
# include <iostream.h>
void main()
{
 int x,y,k,I;
 x =  y + k——I;
 cout << x;
}
```

3. Predict the output

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

void main()
{
  int i=20;
  cout << i << i++ << ++i;
  getch();
}
```

b.
```
# include <iostream.h>
# include <conio.h>
void main()
{
  clrscr();
  int i = 1, a= 3;
  i = a++;
  cout << i;
  getch();
}
```

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

void main()
{
  clrscr();
  int i = 3,x;
  x = i ? i++ : ++i;
  cout << x;
  getch();
}
```

d.
```
# include <iostream.h>
# include <conio.h>
void main()
{
  int z,x = 3, y = 2;
  z = --x + y++;
  cout << z;
  getch();
}
```

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

void main()
{
  clrscr();
  char ch = 'a';
  ch = (ch == 'b') ? ch :' b';
  cout << ch;
  getch();
}
```

4. Evaluate the following C++ expressions

Assume a = 5, b =3, d =1.5, c is integer and f is float.

a. f = a+b/a

b. c = d * a+b

c. x = a++ * d + a;

d. y = a – b++ * —b;

e. (x >= y) ||(!(z==y) && (z < x)) where

- ✓ x = 10, y = 5, z = 11 (all are integers)
- ✓ x = 10, y = 10, z = 10
- ✓ x = 9, y = 10, z = 2

5. Write the C++ equivalent expressions using the conditional operator. Where

- ✓ f = 0.5 if x = 30, otherwise f = 5
- ✓ f n = 0.9 if x >= 60, otherwise .7

6. What are pointer variables ?

7. Write a declarative statement to declare ‘name‘ as a pointer variable that stores the address pointing to character data type.