Discipline Courses-I
Semester-I
Paper: Programming Fundamentals
Unit-I
Lesson: Arithmetic and Relational Expressions and Data Types

Lesson Developer: Tarang Jain
College/Department: Moti Lal Nehru College, University of Delhi

Table of Contents

- Chapter 3: Arithmetic and Relational Expressions
  - Introduction
  - 3.1 Expressions
  - 3.2: Arithmetic Operators
    - 3.2.1: Basic Operators
    - 3.2.2: Increment/ Decrement Operators
  - 3.3 Relational Operators
  - 3.4 Hierarchy and Associativity of Operations
  - 3.5 The "Temperature Converter" Program

- Chapter 4: Data Types
  - 4.1 Numeric Data Types
  - 4.2 Character and String Data Type

(# void, integer, char, floating point, logical data in C++)

- Chapter 5: Type Conversions
  - 5.1 Automatic conversions
  - 5.2: Type Cast
  - Summary
  - Exercises
  - Glossary
  - References

Introduction

Welcome to Chapter 3! In the previous chapter, you learned about variables and constants. In this chapter, you will learn about arithmetic and relational operators. You will also learn about the hierarchy of these operators and how to write a "Temperature Converter" program using them.
This will help you understand the concept of operators used in a C++ program.

Learning Objectives

After reading this chapter you should be able to:

1. Recognize arithmetic and relational expressions in a C++ program.
   Define what an expression is.
   Discuss the basic arithmetic operators used in C++.
   Use of increment/decrement operators.
   Discriminate between relational operators and arithmetic operators.
   Write a simple C++ program using arithmetic and relational operators.

3.1 Expressions

If we take the English dictionary meaning of the word "expression", it says "word, statement" etc. Approximately similar is the meaning of the word "expression" in a computer language.

3.1.1 Expression

An ***expression*** is any valid combination of operators, constants, and variables arranged as per the rules of the language. The rules of formation of an expression are:

1. A signed or unsigned constant or variable is an expression.
2. An expression connected by an operator to a variable or a constant is an expression.
3. Two expressions connected by an operator is also an expression.
4. Two operators cannot occur in continuation.

Examples of expressions are:

```
67                      //arithmetic expression
34 + 59 * 7             //arithmetic expression
'x'                     //arithmetic expression
A                       //arithmetic expression
a + b – 7 * c           //arithmetic expression
&a                      //arithmetic expression
a>b && c==10            //Boolean expression
a>>2                    //Boolean expression
```

An expression, that involves arithmetic operators and produces a numerical value, is known as an ***Arithmetic expression***. An expression, that involves relational/logical operators and produces a Boolean value (true/false), is known as a ***Boolean expression***.

Value addition: Did You Know
Heading text: Expression
Body text: • In C++, five arithmetic operators are allowed for arithmetical computations. These are +,-,*,/ and %. Also six relational operators are allowed for comparison. These are >=,<=,>,<,== and !=. These operators are discussed in detail later on. • An **operand** is the value on which an operator works on. For example: in the expression *a + b * c* a, b and c are operands and + and * operators are working on them. • An operator can be a unary operator or a binary operator. For example in the expression, - a + b * c + and * are binary operators and – is a unary operator. • The syntax for a binary operator is : <operand> <binary operator> <operand> • The syntax for a unary operator is : <operand> <unary operator> or <unary operator> <operand>

Value addition: Note

Heading text: Expression

Body text:

1. Sometimes, when we work on operands having unary operator – (minus), two operators comes in continuation. You have to use parenthesis to avoid consecutive operators. Otherwise the compiler will show a syntax error. For example,

```
a/b + -b*c –d       // Incorrect, as + and – are consecutive

a/b+(-b*c –d)       // Correct
```

   Here parenthesis are used to separate the binary operator + and the unary operator -.
2. An expression may also use combinations of arithmetical and relational expressions. Such expressions are known as ***compound*** expressions. For example:

```
a/b+(-b*c –d) + a>b -c==d
```

   The above expression contains an arithmetic expression as well as a relational expression.

3.2 Arithmetic Operators

3.2.1 Basic Operators

There are five basic arithmetic operators that can be used in C++. These are + (addition), - (subtraction), * (multiplication), / (division), and % (modulus or remainder). The operators + and – can act as *unary* as well as *binary* operators. The operators *, /, and % are binary operators. The operands can be integer, floating-point or character type variables or constants. Second operand must be a non-zero operand for division and modulus operator. Because if the second operand is a zero value, then division by zero means infinite, and infinite is not allowed in any computer language. In any computer programming language, everything is finite. It may be a very big value, but it is finite. Let us have some examples on these arithmetic operators:

Let the integer variable a=9, and the integer variable b=4. Then:

```
a+b=13
a-b=5
a*b=36
a/b=2
a%b=1
+a=13        //unary operator
-a=-13       //unary operator
```

Let a=14.5 and b=2.0 then:

```
a+b=16.5
```

a-b=12.5
a*b=29.0
a/b=7.25

In this way arithmetical operators can be used for arithmetical computations.

Value addition: Some More Information
Heading text: Arithmetic Operators
Body text: 1. Arithmetic operators can be applied on characters also, e.g. let ch='q', then ch=ch-1 will assign 'p' to ch. Similarly, let i=24; then i=i+'B' will assign 90 to integer variable i. Here ASCII (American Standard Code for Information Interchange) values of the characters are used for computation. 2. The division of an integer by another integer always results in an integer value. The decimal part will be truncated. For example: 5/2=2 17/4=4 and so on 3. If both or one of the operands in a division is floating point, the result is always a floating point number. For example: 5.0/2=2.5 17/4.0=4.25 3.0/2.0=1.5 4. The modulus operator produces remainder of an integer division. This operator can not be used with floating point numbers. For example: 5%2=1 17%3=2 and so on 5.25%2=? //not allowed

3.2.2 Increment/ Decrement Operators

The operator ++ is known as increment operator and -- is known as decrement operator. The increment operator is used to increase the value of a variable by 1 and decrement operator is used to decrease the value of a variable by 1. Therefore

```
x=x+1
```

is equivalent to

```
x++
  or
++x
```

and

```
x=x-1
```

is equivalent to

```
--x
  or
x--
```

let us say that initially the value of integer variable x is 5. Then after the execution of the statement

x++

the value of the variable x will become 6.
The increment and decrement operators can be applied before the variable (prefix) or after the variable (postfix) to a variable as shown above. There is a difference in the prefix or postfix form of these operators when these are used in an expression. When an increment or decrement operator precedes its operand (prefix operator), the increment or decrement operation is performed before obtaining the value of the operand to be used in the expression. However, if the operator follows its operand (postfix operator), the value of the operand is obtained before incrementing or decrementing it, that is, the value is increased or decreased after using it in the expression. For example:

y=x++
is equivalent to
y=x followed by x=x+1

While y=++x
is equivalent to
x=x+1 followed by y=x

Same is the case with the decrement operator. It means

y=x--
is equivalent to
y=x followed by x=x-1

While y=--x
is equivalent to
x=x-1 followed by y=x

Value addition: Note

Heading text: How to remember

Body text:

To remember whether the increment/ decrement will be done before/ after evaluating the expression, keep in mind the position of the operator.

1. If the operator is before the operand, then the increment/ decrement will be done first and then new value will be used in the expression evaluation.
2. If the operator is after the operand, then the expression will be evaluated first and then the increment/ decrement will be done.

Let us write a program to do some arithmetical computations.

```
/* 3.1 write a program to do some simple arithmetic calculations */

#include <iostream>                          //header file
using namespace std;
int main()                                   // main function
{
        int a=17,b=3,c;       //variables declaration
        float a1=17.5,b1=3.1,c1;

/*operators applied on integer values */
        cout<<"a+b="<<a+b<<"\ta-b="<<a-b<<"\ta*b="<<a*b<<endl;
        cout<<"a/b="<<a/b<<"\ta%b="<<a%b<<endl;
        ++a;                          //increment operator
        cout<<"++a="<<a<<endl;
        b--;                          //decrement operator
        cout<<"b--="<<b<<endl;
        c=++a - b--;
        cout<<"a="<<a<<"\tb="<<b<<"\tc="<<c<<endl;

/*operators applied on floating-point values*/
        cout<<"a1+b1="<<a1+b1<<"\ta1-b1="<<a1-b1;
        cout<<"\ta1*b1="<<a1*b1<<endl;
        cout<<"a1/b1="<<a1/b1<<endl;

/*modulus operator is not allowed on floating-point numbers*/
        //cout<<"\ta1%b1="<<a1%b1<<endl;    //syntax error

        ++a1;
        cout<<"++a1="<<a1<<endl;
        b1--;
        cout<<"b1--="<<b1<<endl;
        c1=++a1 - b1--;
        cout<<"a1="<<a1<<"\tb1="<<b1<<"\tc1="<<c1<<endl;

        return(0);
}
```

The output of the above program would be:

```
a+b=20      a-b=14      a*b=51
a/b=5       a%b=2
++a=18
b--=2
a=19        b=1         c=17
a1+b1=20.6  a1-b1=14.4  a1*b1=54.25
a1/b1=5.64516
++a1=18.5
b1--=2.1
a1=19.5     b1=1.1      c1=17.4
```

3.3 Relational Operators

A relational operator is used to compare two values. When we compare values, the result is in the form of "yes", or "no". Same is the case here. When we compare two values using relational operators, our result comes as a Boolean value (either true or false). Six relational operators can be used in C++. These are shown in the following table 3.1:

Operator	Description	Example	Explanation
==	equal to	a==b	Returns true only if the values of a and b are equal, otherwise false is returned
!=	Not equal to	a!=b	Returns true only if the values of a and b are not equal, otherwise false is returned
>	Greater than	a>b	Returns true only if the value of a is greater than the value of b, otherwise false is returned
<	Less than	a<b	Returns true only if the value of a is less than the value of b, otherwise false is returned
>=	Greater than or equal to	a>=b	Returns true only if the value of a is greater than or equal to the value of b, otherwise false is returned
<=	Less than or equal to	a<=b	Returns true only if the value of a is less than or equal to the value of b, otherwise false is returned

Table 3.1: Relational operators allowed in C++

Let us write a simple program to use these relational operators for comparison and see how it works:

```
/* 3.2 Write a program to use some relational operators for comparison */

#include <iostream>                          //header file
using namespace std;

int main()                                          // main function
{
        int a=17,b=3,c,c1;   //variables declaration
        float a1=17.5,b1=3.1;

        c=a>b;               //integer operands
        c1=a<b;
        cout<<"a>b="<<c<<"\ta<b="<<c1<<endl;

        c=a1>=b1;            //floating-point operands
        c1=a1<=b1;
        cout<<"a1>=b1="<<c<<"\ta1<=b1="<<c1<<endl;

        c=a1==b1;
```

```
            c1=a!=b;
            cout<<"a1==b1="<<c<<"\ta!=b="<<c1<<endl;

            return(0);
}
```

The output of the above program would be:

```
a>b=1           a<b=0
a1>=b1=1        a1<=b1=0
a1==b1=0        a!=b=1
```

The output of the above program comes as '0' and '1' where '0' represents false and '1' represents true in C++.

Value addition: Assignment Operators

Heading text: Operators

Body text:

In C++, =, +=, -=, *=, /=, %= are used as assignmest operators. The self explanatory examples for these operators are:

```
A=5;
B=56;
C=A+B;
```

A+=B is equivalent to *A=A+B.*
A-=B is equivalent to *A=A-B*.
A=B* is equivalent to *A=A*B*.
A/=B is equivalent to *A=A/B*.
A%=B is equivalent to *A=A%B*.

3.4 Hierarchy and Associativity of Operators

3.4.1 Hierarchy of Operators

The order, in which the operators will be evaluated in an expression, is known as the hierarchy. The following table shows the hierarchy of the operators discussed till now. The operators with the same level of precedence are given in the same row. An operator with higher level of precedence will be evaluated before an operator with lower precedence. The order of precedence can be changed using parenthesis.

Highest precedence	()
Unary operators	+ - ! ++ --
Multiplicative operators	* / %
Additive operators	+ -
Relational operators	< <= > >=
Equality operators	== !=
Logical operators	&& ||
Assignment operators	= += -= *= /=
Lowest precedence	

Examples:

a+b*c-d is equivalent to a+ (b*c) -d i.e. first multiplication will be done and then addition and subtraction will be done because precedence of multiplication is higher than that of addition or subtraction. To override this order of precedence, we have to use parenthesis as shown below:

(a+b)*(c-d).

Now multiplication will be done after addition and subtraction.

3.4.2 Associativity of Operators

Associativity of operators means whether the operators would be evaluated from *left to right* or form *right to left*. All the operators are associated from left to right except the *unary* operators and the *assignment* operators. For example:

x=a+b*c

Here, right hand side expression will be evaluated first and then the result will be assigned to variable x.

3.6 The "Temperature-Converter" program

Now that you know about operators, Let us write a simple program that uses arithmetic operators. The program will convert a temperature given in Fahrenheit to Celsius.

```
/* Write a program to convert temperature from Fahrenheit to Celsius */

#include <iostream>                                  //header file
using namespace std;
int main()                                           //main function
{
      float tempFarh, tempCel;                       //variable declaration
      cout<<"Enter temperature in fahrenheit: ";
      cin>> tempFarh;                                //input statement
      tempCel=5.0/9 *( tempFarh-32);                    // use of arithmetic expression
      cout<<"Temperature in Celsius is "<<tempCel;
      return 0;
}
```

The output of the above program would be:

```
Enter temperature in fahrenheit: 99
Temperature in Celsius is 37.2222
```

In this program, we have declared two floating type variables *tempFarh* and *tempCel*. We receive the value of temperature in Fahrenheit from the user using *cin.* Now look at the next statement where we have calculated the temperature in Celsius. Since the precedence of * is higher than that of – operator, so we used brackets - () to override this. Also we have used 5.0 in place of 5. If we use 5, then 5/9 will become 0 due to *integer division.* In the next statement, we display the result to the user using *cout* and then we return from the function to the operating system.

•

Chapter 4. Data Types

Welcome to Chapter 4! In the previous chapter, you have learned about arithmetic and relational operators. In this chapter, you will learn about numeric data type, character data type, and a little about string data type. You will also learn about the logical data type *bool*. To write a program in C++, you have to use variables. Before using them, you have to specify their types. In this chapter we are going to discuss some of the basic data types.

Learning Objectives

After reading this chapter you should be able to know about the basic data types available and used in a C++ program.

1. What do you mean by a data type.
2. A detailed discussion on basic data types.
3. Modification of these basic data types.
4. Why and when to modify them?
5. An idea of strings.
6. A new data type used in C++ i.e. *bool*.

4. Data Types

Data is a general term used to denote any or all the facts, numbers, symbols and letters that refer to or describe an idea, situation, or condition. Every program is written to solve a problem. By problem we mean that we have some data and we want to get some results after some processing on it. Now this data can be of any type like it may be numeric (numbers), alphabetic, alphanumeric, list, matrix etc. Now whenever we write a program in C++, we have to specify which type of data is being used. For this purpose, we have many categories of data type in C++ as shown in the following figure 4.1:

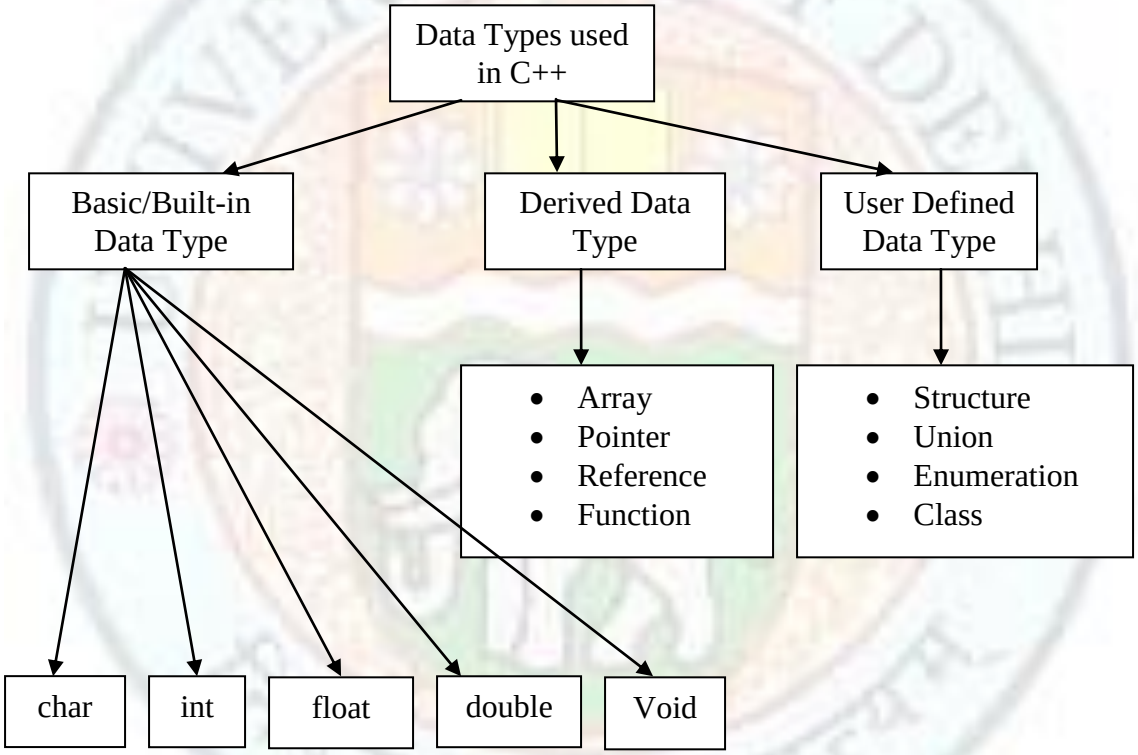

Figure 4.1: Data types used in C++

4.1 Basic/Built-in Data Types

As it is clear from the figure 4.1, there are three types of data used in C++. These are user-defined data types, basic data types, and derived data types. User-defined data types and derives data types are based on basic data types. The basic data types are further divided into three categories. These are integral type, void and floating type. Integral type is further divided into two categories. These are int, char. And floating type is also divided into two categories. These are float and double. The basic data types are also known as built-in data types.

a. **Character Type (char):** This data type is discussed later.

b. **Integerl Type (int):** It is a whole number without a decimal point. These numbers are used for counting. Some examples of valid integers are:

476
9832
-36

Normally an integer can hold numbers from -32768 to +32767 (i.e. -2^{16-1} to $+2^{16-1} - 1$).

c. **Floating-point Type (float):** A floating-point number has a decimal point. These numbers are used for measuring quantities. Some valid examples of floating point numbers are:

123.456
-897.34
234.0003

A floating-point number can hold numbers from 10^{-38} to 10^{38}. The precision of the digits is normally 6 digits. By precision we mean the number of digits allowed after decimal point.

d. **Floating-point Type (double):** In double type of data type, the precision of digits will increase by double-fold and its range is also very high. Otherwise, it is same as *float* type of data type. Here it can hold numbers from 10^{-308} to 10^{+308} with about 15 digits of precision. For example:

123456789.456783527

e. void: The type *void* either explicitly declares a function as returning no value or creates generic pointers. It also indicates that a function contains an empty argument list. It was introduced in *ANSI C*.

```
void function-name (void);
```

Here it shows that the function *function-name* is returning no value and also that this function does not contain any arguments. This concept will be clearer to you when you will study about functions.

```
void *ptr;
```

Here *ptr* has become a generic pointer. A generic pointer can be assigned a pointer value of any basic data type, but it may not be dereferenced. For example:

```
int *a;
```

```
ptr=a;
```

here an *int* pointer is assigned to *void* pointer. But

```
*a=*ptr;
```

is illegal as it would not make sense to dereference an integer pointer to a void value.

Value addition: Did You Know
Heading text: void

Body text:
Assigning any pointer to a *void* pointer without using a cast is allowed in both C++ and ANSI C. In ANSI C, we can also assign a *void* pointer to a non-void pointer without using a cast to non-void pointer type. This is not allowed in C++. For example:
`void *ptr1;`
`char *ptr2;`
`ptr2=ptr1;`
are all valid statements in ANSI C but not in C++. A *void* can not be directly assigned to other type pointers in C++. We have to use a cast operator to do so. For example:
`ptr2=(char*)ptr1;`

Modifying the basic data types:
With the exception of *void*, the basic data types have several modifiers preceding them to serve the needs in various situations. A modifier is used to alter the meaning of the base type to fit various situations more precisely. The list of modifiers is:

signed
unsigned
long
short

The modifiers *unsigned* and *signed* can be applied to char data types. All the above four modifiers can be applied to integer data type. The modifier *long* can be applied to *double* data type. Table 4.1 shows all the valid data type combinations. It also gives the range of these data types and the number bytes used by them. As it is clear from the table, the use of *signed* is allowed with integer type data but it is redundant because the default int declaration is same as *signed int*. Similarly, *signed char* is same as *char,* as by default a character is signed character.

Type	Bytes/Bits	Range
char	1/8	-128 to 127
signed char	1/8	-128 to 127
unsigned char	1/8	0 to 255
int	2/16	-32768 to 32767
signed int	2/16	-32768 to 32767
unsigned int	2/16	0 to 65535
short int	2/16	-32768 to 32767
signed short int	2/16	-32768 to 32767
unsigned short int	2/16	0 to 65535
long int	4/32	-2,14,74,83,648 to 2,14,74,83,647
signed long int	4/32	-2,14,74,83,648 to 2,14,74,83,647
unsigned long int	4/32	0 to 4,29,49,67,295
float	4/32	3.4e-38 to 3.4e+38 (precision 6 digits)
double	8/64	1.7e-308 to 1.7e+308 (precision 10 digits)
long double	10/80	3.4e-4932 to 3.4e4932 (precision 10 digits)

Table 4.1: Data types and their range

Value addition: Did You Know

Heading text: Basic Data Types

Body text:

1. Standard C++ does not specify a minimum range or size for the basic data types. Instead, it simply states that they must meet certain requirements. For example, Standard C++ states that an *int* will have the natural size as given by the architecture of the execution environment. Each C++ compiler specifies the size and range of the basic types in the header file <*climits*>.
2. The range of *float* and *double* will depend upon the method used to represent the floating-point numbers.

Value addition: FAQ

Heading text: Modifiers Used on Basic Data Types

Body text:

When a type modifier is used by itself i.e. when it does not precede a basic data type, then by default *int* is assumed. For example:

signed is equivalent to *signed int*
unsigned is equivalent to *unsigned int*
short is equivalent to *short int*
long is equivalent to long int

4.2 Character and String Data Type

Character Type (char): It is a non numeric data type consisting of single alphanumeric character. Some examples of valid characters are:

't'
'B'
'#'
'+'
'5'

Value addition: Note
Heading text: char
Body text: • 5 and '5' are different types of variables. 5 is an integer while '5' is a character. • Values of type *char* are generally used to hold values defined by the ASCII character set.

String: A ***string*** is a sequence of characters. A *char* type data include only one character, but many times we need to use more than one character. In that case we use string type of data. A string is a group of characters of any length. When we enclose a group of characters in double quotation marks, then it is known as a "*string literal*". For example "computer" is a literal. The strings are stored as an array of characters. (The concept of array is discussed later on.) A string is always terminated with a null character '\0'. The ASCII value of this character is 0. This null character will increase the length of the string by 1. Therefore, the effective size of an array of characters is one more than the size of the string it can hold. In ANSI C++, there is a new class known as **string**. (The concept of class is discussed later on.) For using the sting class, we have to use ***#include <string>*** in that program. It is a very big class and includes many functions, constructors and operators to create, read, modify, compare, or display strings.
This concept will be discussed later in detail.

Value addition: New data type bool
Heading text: Basic data types

Body text:
In addition to the above data types, there is one more built-in data type available in C++. This is known as **bool.** In this data type, only two values are allowed. These values/keywords are **true** or **false.** In C++, automatic conversion of *bool* type data takes place to integers. Here, *true* is converted to 1 and *false* is converted to 0. reverse is also true i.e. any non-zero value is considered as *true* and zero is taken as *false*.
 	
bool b;
b=false;
bool b1=true;

```
int i=3+8*b1-false+b-b1;   // i=10 as b1 is 1 and b is 0
```

Value addition: Basic Information

Heading text: Derived Data Types

Body text:

There are four Types of derived data types. These are array, pointer, function and reference.

1. **Array:** When we have a lot of variables, it is not possible to learn the name of all of these variables. If all these values are of the same type then we can have a single name for these variables using *array*. The values of these different variables are differentiated using index. For example, suppose in a class there are 30 students and we want to keep roll no of all these students. Then instead of having 30 different variables, we can have a single variable *rollNo* of array type with 30 different indexes starting from rollNo[0] to rollNo[29]. This array type variable *rollNo* is known as a single dimensional array. Similarly we can have multi-dimensional array.

   int rollNo[30];

   This concept of array is discussed in detail later on.
2. **Pointer:** Pointer is a different kind of variable which contains the address of another variable. For example:

   ```
   Int I,*ptr;
   I=254;
   ptr=&I;
   ```

   Here ptr is a variable of pointer type which can hold the address of an integer variable I. Using pointer variable, we can return more than one value from a function. This concept is also discussed in detail later on.
3. **Function**: It is also known as a *subprogram*. It has a return type and it may have zero or more arguments in its bracket. You have already used *main()* function in your programs. A function can return only one value by its name. To return more than one value from a function, pointers or reference variables has to be used. This concept is also discussed in detail later on.
4. **Reference:** This is a new kind of variable introduced in C++. It provides an alias (alternative name) for a previously defined variable. Using reference variable, we can return more than one value from functions. For example:

   ```
   double d=234.56;
   double &d1=d;
   ```

Value addition: Basic Information

Heading text: User-defined Data Types

Body text:

There are four Types of user-defined data types. These are structure, union, enumeration and class.

1. Structure: In case of an array we can have the same name for more than one variable. But all of these variables has to be of the same type. But sometimes we need to store information in a variable which is not homogenous but heterogeneous. For example, if we want to keep information about a student, then we need his name, address, roll no, class, marks etc. now marks can not be string type, and name can not be integer type. To store these kind of information, we use *structure* data type. The syntax for it is:

```
struct <structure name>
{
    <data type> <variable1>;
    <data type> <variable2>;
    <data type> <variable3>;
    …
}<variables>;
```

For example:

```
struct student
{
    char rollNo[10];
    char name[15];
    int age;
    int marks[3];
}stud1,stud2;
```

You will learn about it later on in detail.

2. Union: It has the same syntax as that of structure data type. It is useful when we want to decrease the requirement of memory. In a program, all the variables are not required at a single time. Sometimes we use variables, in which if one is required, then other is not required and vice versa. In that case *union* is useful. The syntax for it is:

```
union <union name>
{
    <data type> <variable1>;
    <data type> <variable2>;
    <data type> <variable3>;
    …
}<variables>;
```

For example:

```
union unintchar
{
    char rollNo[2];
    int age;
}un1;
```

Here the variables *rollNo[2]* and *age* will use same memory locations.

You will learn about it later on in detail.

3. Enumeration: This data type is used to attach names to numbers. By doing so, the clearability and understandability of the program increases. The syntax for it is:

```
enum <variable name>{
```

<member/s >
};

For example,

enum sex{ male, female};

Here male=0 and female=1 and so on.

enum colour{ red, green=5, blue, pink=10, orange};

You will also learn about it later on.

3. **Class: It is a group of objects. A class contains the member variables, and the member functions working on these member variables. The syntax for a class is:**

```
class <class name>
{
   private:
        member variables
        member functions
   public:
        member variables
        member functions
}<object-name>;
```

Again, it will be discussed in detail later on.

Chapter 5. Type Conversions

Welcome to Chapter 5! In the previous chapter, you have learned about data types (numeric, character, string and bool). In this chapter, you will learn about their conversions i.e. how one data type can be converted into others.
This will help you understand the concept of casting used in a C++ program.

Learning Objectives

After reading this chapter you should be able to know about casting used in a C++ program.

1.1 What is casting?
1.2 In which cases casting is done automatically?
1.3 When the programmer has to type cast a variable himself/herself?
1.4 Write some simple C++ programs.

5. Type Conversions

In C++ language, the compatibility of variables is very much required. When variables of one type are mixed with variables of another type, then some type conversions will occur. Now, these conversions can be automatic or may have to be done by the programmer. Otherwise, the results may get affected. C++ is very strict with respect to type compatibility.

5.1 Automatic Type Conversions

In case of assignment statements, when you mix different types of variables, automatic type conversions take place. Here the result we get, after evaluating the right hand side expression, will be automatically converted to the type of the variable on the left hand side. Sometimes, some of the information will be lost due to these conversions and you may not get the exact results as required by you. So always take care of these types of automatic conversions. Let us try to understand this concept using a simple program.

```
/* 5.1 write a program in C++ where automatic conversions of variables is done */

#include <iostream>                    //header file
using namespace std;

void main()                            //main function
{
        int p=637;                     //variable declarations
        char ch='a';
        float f=387.29;
        double d=983.123456789;

        cout<<"The exact values stored in these variables are ";
        cout<<p<<'\t'<<ch<<'\t'<<f<<'\t'<<d<<'\n';

        ch=p;  //integer is stored to character

        p=f;    //float is stored to integer

        f=d;    //double is stored to float

        cout<<" The new values are ";
        cout<<p<<'\t'<<ch<<'\t'<<f<<'\t'<<d<<'\n';

        //when small values are moved to big variables
```

```
                d=f;    //float is moved to double

                f=p;    //integer is moved to float

                p=ch;  //character is moved to integer

                cout<<" Now the values are ";
                cout<<p<<'\t'<<ch<<'\t'<<f<<'\t'<<d<<'\n';
}
```

The output will be:

```
The exact values stored in these variables are   63    a     387.29   983.123

The new values are   387    ?       983.123       983.123

Now the values are   63      ?       387    983.123
```

As it is clear from the output, when an integer value is stored in a character value, its ASCII equivalent will be given to you. When a float or double value is moved to an integer, it will be truncated. Also, when an integer is moved to float or double; or a float is moved to double, no additional information is added.

Value addition: FAQ

Heading text: integer to char

Body text:

If the integer value is between -128 to 127, then you will get the exact values for p and ch as shown in the above program. Otherwise, these will be changed. For example, if :

```
P=637;

Ch='a'; integer to char

Then after conversion, i.e.

Ch=P;

The output of Ch will be

{
```

The following **Table 5.1** summarizes the assignment type conversions:

Expression type	Target type	Possible info loss
char	signed char	If value>127, target is negative
short int	char	If value is between -128 to 127, then no loss
int	char	If value is between -128 to 127, then no loss
long int	char	If value is between -128 to 127, then no loss
int	short int	None
long int	int	If value is between -32768 to 32767, then no loss
float	int	Fractional part is truncated. If integer part is between -32768 to 32767, then integer part will be same.
double	float	Precision
long double	double	Precision

Table 5.1: Assignment Type Conversions

Automatic conversion also takes place while evaluating the expressions. Here if one operand is integer and another is float or double, then automatically the integer value will be converted to float or double for that expression. That means, the automatic conversion will move in upward direction. For example:

Let int p;
float f;
double d;
then
p*f will give float value.
P*d will give double value.
f/d will give double value and so on.

5.2 Type Casts

As you know by now when an integer is divided by an integer, an integer will be given as output. For example, 2/3 will give 0 as output (as the fractional part will be truncated). Also 7/3 will give 2 as output. In integer division, only the quotient part is considered and the remainder part is discarded. Sometimes we can not change the type of the operand for the whole program, but still we need the fractional part of the expression to get the exact result. If we could be able to change the type of one of the operand for the current expression, then our work will be done. For this purpose we have to cast the operand. By type casting we can change the type of the operand for a particular expression.

C++ is very strict with regard to type compatibility. For instance, *int, short int, unsigned int, long int etc*. all these are different. In case of function overloading, this type of compatibility matters. So to avoid this kind of problem we have to cast the values so that a particular function could have been called. (This concept will be clear to you when you will study function overloading later.)

C++ provides this facility of type casting a variable or an expression. The syntax for type casting is:

```
(type-name) expression
type-name (expression)
```

For example:

```
(float) a
float (a)
```

Both types of syntax are valid in C++. Let us discuss it using a program.

```
/* Program 5.2: How to change the type of a variable for a particular
expression */

#include <iostream>                 //header file
using namespace std;

void main()                         //main function
{
        int p,q,r;                  //variable declarations
        float f;

        p=5;
        q=22;
        f=29.45;

        cout<<" the values are ";
```

```
        cout<<q/p<<'\t'<<f/q<<'\t'<<f/p<<'\t'<<(float)q/p<<'\t'<<q/float(
p)<<'\n';                //casting of variable p and q
}
```

The output of the program would be:

the values are 4 1.33864 5.89 4.4 4.4

Summary

- An *expression* is any valid combination of operators, constants, and variables.
- Arithmetic operators that can be used to perform arithmetic calculations are +,-,*,/ and %.
- ++ and -- are increment and decrement operators respectively.
- The relational operators that can be used to compare values are >, >=, <, <=, ==, !=.
- Hierarchy means order of precedence of operators. It determines the order in which operators will be evaluated in an expression.
- Before using a variable or a constant, its type has to be declared which will specify the range/values allowed for that variable or constant.
- There are 5 types of basic/built-in data types. These are char, int, float, double, and void.
- Basic data types can not be divided further.
- A new basic data type is added in C++. This is known as *bool*. It is logical data type.
- A string is a combination of characters.
- The type of a variable can be changed in two ways: Automatically or by casting.
- In assignment statements, automatic type conversions take place.
- Automatic conversions also take place in arithmetical expressions.
- Sometimes, we have to cast a variable to change its type to get exact results in arithmetic expressions.

Exercises

3.1 Differentiate between the following:

b. Arithmetic operators and Relational operators

c. Prefix and Postfix increment operators

3.2 Write a program in C++ to convert temperature form Celsius to Fahrenheit.

3.3 Write a program in C++ to calculate simple interest. You are given the values of principal, rate and time.

3.4 True/false

a. Modulus operator can be applied on floating-point numbers.

b. Relational operators are used for comparisons.

c. The arithmetical operators * and + have same precedence.

3.5 Write the C++ equivalent expression for the following:

a. b^2 -4ac

b. $x = 1-x^2/2+x^4/4-x^6/6$

3.6 What will be displayed when the following code is executed?

```
#include <iostream>
using namespace std;
int main()
{
        int i=0;
        i=400*400/400;
        cout<<i;
        return 0;
}
```

4.1 Differentiate between the following:

a. integral type data and floating-point data

b. double and long double

c. int, signed int and unsigned int

d. character type data and string type data.

4.2 Is it true that an unsigned int is twice as large as the signed int? Why?

4.3 Can we assign a void pointer to an int type pointer? Why?

4.4 Write a short note on void data type.

4.5 What is the use of modifiers on basic data types?

4.6 What will be the output of the following program:

```
#include <iostream>
using namespace std;
int main ()
{
        cout<<7+ 7/ 7.0;
        return (0);
}
```

5.1 Write a program in C++ to convert temperature form Fahrenheit to Celsius using type casting of integer variable to float.

5.2 State true/false

a.) In an assignment expression, the type casting of variable on the left hand side can be done.

b.) There is never a loss of information when an integer variable is divided by another integer.

c.) By type casting an expression, the accuracy of the result can be increased.

5.3 What will be the output of the following program?

```
#include <iostream>
using namespace std;
int main ()
{
        float f;
        f=5/2*float(7)/(int)3.5;
        cout<<f;
        return (0);
}
```

5.4 Sometimes when an integer value is moved to a character variable and then that character value is moved back to the integer variable, we get negative value. Why?

5.5 What will be displayed when the following code is executed?

```
#include <iostream>
using namespace std;
int main ()
{
        int i=2, j=5;
```

```
        float f1,f2;
        f1=j/i;
        f2=float(j)/i;
        cout<<f1<<f2;
        return 0;
}
```

Glossary

Arithmetic operator – An operator used for arithmetic calculations. e.g. +,-,*,/,%

Automatic conversions- When the type of the expression changes automatically, it is called automatic conversion.

Basic/Built-in data type - Basic data types are not composed of any other data type. These can be used in their existing format only.

Binary operator – An operator that has two operands.

Derived data type– Derived data types can be further divided in to basic data types.

Expression - A valid combination of operators, constants, and variables.

Logical data type- Logical data types are used when the output is logical i.e. either true or false.

Operand- The value on which an operator works is known as an operand.

Postfix operator – An operator comes after operand

Prefix operator – An operator comes before operand

Relational operator – An operator used for comparison. e.g. <, >=, != etc.

String – A string is a combination of more than one character.

Type Casting- It is to change the type of the expression to increase the accuracy of the result.

Type Conversions– To change the type of the value is known as type conversion.

User defined data types – User defined data types are constructed using basic and derived data types.

Unary operator – An operator that has only one operand.

References

1. A.K.Sharma, Introductory Computer Science (Volume II), Dhanpat Rai Pub.
2. B. A. Forouzan and R. F. Gilberg, Computer Science, A Structured Approach using C++, Cengage Learning, 2004
3. E. Balaguruswamy, Object Oriented Programming with C++, Tata Mcgraw Hill
4. H. Schildt, C++: The Complete Reference Book, Tata Mcgraw Hill