CHAPTER 3: LANGUAGE STRUCTURES OF OOP

Basic Structure of C++ Program

As C++ is a programming language so it follows a predefined structure. The program is divided into many sections, it is important to know the need of every section. The easiest way to understand the basic structure of c++ program is by writing a program. The basic C++ program is as follows:

```
//simple c++ program

#include<iostream> // header file included

using namespace std;

int main()

{

int a=10,b=34;

cout<<"simple c++ program \n";  // c++ statement

cout<<"hello world";

cout<<a<<b;

return 0;   // returning no errors

}
```


The basic structure of c++ program mentioned above can be divided into following sections:

- **Documentation Section :** This section comprises of comments. As the name suggests, this section is used to improve the readability and understanding of the program.// (Double Slash) represents comments in C++ program. Comments can be of single line or multiple lines. Double Slash comments are used to represent single line comments. For multiple line comment, you can begin with /* and end with */. For example :

/* Text line number 1

Text line number 2

Text line number 3 */

In the above C++ program **//simple c++ program** represents single line comment.

- **Linking and Directives Section :** The program written above begins with #include<iostream>. <iostream> represents header file which includes the functionalities of predefined functions. In linking section, the compiler in-built functions such as cout<<, cin>> etc are linked with INCLUDE subdirectory's header file <iostream>. The '#' symbols tells about "address to" or "link to". Iostream is input/output stream which includes declarations of standard input-output library in c++.
- **main() Section :** This is the section in which the program coding is written. Basically, it acts as a container for c++ program. The execution of the c++ program begins with main() function and it is independent of the location of main() function in the program. main() is a function as represented by parenthesis "()". This is because it is a function declaration. The body of the main() function can be found right after these parenthesis, the body is enclosed in braces "{ }".
- **Body of main() Section :** The body of the main() function begins with "{".
    - Local Variable Declaration : In this the variables which are used in the body o f the main() functions are declared. These are called the local variables as their scope is limited within the main() function only, unless they are declared globally outside the main() function. " **int a=10, b=34;**" in the above program represents local variables
    - Statements to Execute : This section includes statements for reading, writing and executing data using I/O functions, library functions, formulas, conditional statements etc. Above written program has many executable statements like **cout<<"simple c++ program \n";**
    - **return 0;** in the above program causes the function to finish and **0** represents that function has been executed with zero errors. This is considered as most usual way to end a C++ program.
    - Finally the body of the main() function ends with "}".
- **Global Declaration Section :** There are certain programs which requires variables that can be used in more than one function, so then the variables can be declared outside the main() function or respective functions. Then those variables become accessible in any of the functions, Hence named as Global Variables as their scope becomes global to the program.
- **User Defined Functions :** There are certain functions that are called by calling statements from the main() function. Every function includes local variable declaration section and executable statement section similar to main program.

One more example which explains the basic structure of c++ program is as follows :

```
1  /* basic example

2     which

3     explains
```

```
4     the structure of c++ program */
5   #include <iostream> // header file include
6  using namespace std;
7   float f=10.2,j=4.5;
8   int main()
9   {
10  int a=10,b=34;
11 cout<<"simple c++ program \n";  // c++ statement
12 cout<<"hello world";
13 cout<<a<<b;
14  return 0;   // returning no errors
15  }
```


- Multiple line comments :

  /* basic example

  which

  explains

  the structure of c++ program */ .

- **float f=10.2, j=4.5;** are global variables which are declared outside the main() function.

The statements written in the above mentioned programs can be written in a single line for example :

```
int main(){int a=10,b=34; cout<<"simple c++ program \n";
cout<<"hello world"; cout<<a<<b; return 0;}
```

- The separation between statements is specified with a semicolon (;). The statements are written in separate lines just to improve the readability of the program.

Features of the Object Oriented programming

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

Header and Source Files and extensions

In this lesson we will talk about some relatively new concepts that I've postponed over and over, until this point, where we can actually use them for a better understanding. We will talk about separating your code into separate **header** and **source** files.

You will finally see what **headers** look like and how we can use them to separate different parts of our code into separate **source files** and **header files**. In this way we can keep our code more organized, by separating different concepts, which will make it much easier to find what we are looking for when trying to modify our program.

There are also much more upsides of doing this, and we will go through each one of them, but let's first take a look at what **headers** really are and look like.

Header files

You've been using the help of a **header** file even from the first program that we wrote in our lessons. If you remember, I've explained, in a very general way, what **headers** are, when we've first encountered the **#include <iostream>** in our programs.

Headers usually have the **.h** extension and contain declarations that we will use in our **source files**. Let's take as an example the **iostream** header file that we include when working with **input-output** in our program. We could have never used **cout** to print to the screen without including the **iostream** header file because we have never declared and defined that identifier anywhere in our program. That is why we are telling the compiler to **include** the **iostream** header file, which actually means that, the compiler will locate and read all the declarations from that header file when it reaches the preprocessor directive, **#include**.

Usually **header files** only contain declarations and do not provide the actual definitions. When we only declare a **function** and do not provide the definition for it, when we are calling that **function** in our program, the **linker** will complain about "**Unresolved Symbols**". So, how does the compiler know where to get the definition for **cout** then?

Well, the **cout** is actually defined in the **standard runtime library** which is automatically **linked** in the **link** process.

What are libraries
Well, simply put, libraries are packages, containers of useful code(**functions**, **objects**) with the purpose of being reused in programs. Now, when you are writing a **library**, you are also writing a **header file** which contains the declarations of the reusable code that exists in that library and you wish to provide to others to include in their own programs.

When you, or others, wish to use the functionality provided by any **library**, you actually do not need the entire source code to be included in your projects. You only need the compiled **library**, (.a, .so, .lib, .dll etc. – depending on the platform you are using) and a **header** file that you will include in your sources when using any of the functionality it provides. Think of **header files** just like you think of a table of contents. It is just a very simple container of declarations, so the compiler will know the minimum it needs about the functionality you are using, when it compiles your code.

When using **libraries**, you do not have to compile the code again, which would be a waste of time, because libraries are always provided as they are, without the need of modifying them. When using a **library** that is not from the **standard runtime**, you will also need to let the compiler know which libraries you wish to include, so it knows where to get the symbols from and link them, once they are used in your program.

For now, we will not use separate **libraries**, but we will get to learn how all of these new concepts work by using separate **source files** and **header files** in our project.

Creating your first separate header and source files
For the purpose of this lesson, we shall create separate header and source files that will contain a couple of basic **functions** that we will use in our **main.cpp**. Let's call that source file **mathPrimer.cpp** and the header file, **mathPrimer.h**. Let's create the file **mathPrimer.h** first, which will contain the following declarations:

mathPrimer.h

```
//  mathPrimer.h
//  ChapterII.HeaderSourceFiles
//
//  Created by Vlad Isan on 20/04/2013.
//  Copyright (c) 2013 INNERBYTE SOLUTIONS LTD. All rights
reserved.
//

#ifndef mathPrimer_h
#define mathPrimer_h

int add(int a, int b);
```

```
int subtract(int a, int b);

#endif
```

First, let's take a look a little at the preprocessor directives in this file. You already know what **#define** is, as we have covered it a little when talking about variables and constants. Let's talk a little about the **conditional compilation**, **#ifdef**, **#ifndef** and **#endif**.
The **conditional compilation** preprocessor directives, tell your compiler what should be compiled or not and under what conditions.
The **#ifdef** preprocessor directive basically checks if something was previously defined using the **#define** preprocessor directive. If this condition is met, then the code between the **#ifdef** and the corresponding **#endif** is compiled, otherwise it is ignored by the compiler.
The same goes for **#ifndef**, and, as you can already imagine, it is the complete opposite of **#ifdef**, and it allows the compiler to check whether a name has not been defined with **#define**.
So, why are we using these preprocessor directives in our **header file**? These preprocessor directives in the **header files** are called **header guards**, and helps us to avoid including the same declaration in multiple times.
This works by skipping the entire contents of the **header file** if it was already included in some other place. In our example, when you first include our header file, **mathPrimer.h**, the **#ifndef** condition is met, because we haven't defined the name **mathPrime_h** by using the **#define** preprocessor directive until now. So, the condition is met and everything inside the **header file** will be compiled, and most importantly, the **mathPrimer_h** will be defined as well, by using the **#define mathPrimer_h** right after the **#ifndef**. Now, when you include the **header file** a second time, the **#ifndef mathPrimer_h** condition will not be met, because we have already defined that name when we included the **header file** in some other place.
By using these **header guards** we are avoiding the complaint we could get from the compiler when declaring the **header** contents twice, or even multiple times.
OK, now let's look at what is declared inside our **mathPrimer.h** header file. We have two function declarations, **add** and **subtract**, both having two integer parameters and an integer return type.
We have to also define our **functions**. Let's create another file called **mathPrimer.cpp**. This file will contain the **function** definitions:

mathPrimer.cpp

```
//
//  mathPrimer.cpp
//  ChapterII.HeaderSourceFiles
//
//  Created by Vlad Isan on 20/04/2013.
//  Copyright (c) 2013 INNERBYTE SOLUTIONS LTD. All rights
reserved.
```

```
#include "mathPrimer.h"

int add(int a, int b) {
    return (a + b);
}

int subtract(int a, int b) {
    return (a - b);
}
```

The first thing we do in this **source file** is to include our **header file**, by using the preprocessor directive **#include "mathPrimer.h"**. Now, when including **header files** from the standard library, you've probably noticed we used angled brackets. The reason is that when including **header files** that come with the compiler, such as standard library **header files**, we use angled brackets, but when including **header files** that we are supplying, we use double quotes**, " "**, which tell the compiler to look for the **header file**, by first searching for it in the current directory where the **source files** are contained in.

Please note that when we only have declarations in the **header files**, we do not need to include the **header file** when defining the **functions**, as we do in our case. We have to only include it when we are calling the **functions** declared in it. But, as we already are using **header guards**, it is a good practice to include it in here as well, because **header files** can also contain **constants** which could be used in here as well. As an example, we can declare a **constant** in our **header file** to hold the value of **PI**, and use that **constant** when defining the other **functions**, in our **mathPrimer.cpp**.

As you can notice, we are also defining the functions **add** and **subtract** which we are going to use in our **main** function.

Now, let's take a look at our **main.cpp:**

```
//  main.cpp
//  ChapterII.HeaderSourceFiles
//
//  Created by Vlad Isan on 20/04/2013.
//  Copyright (c) 2013 INNERBYTE SOLUTIONS LTD. All rights
reserved.
//

#include <iostream>
#include "mathPrimer.h" /* including our header file */

using namespace std;

int main()
{

    /* calling our functions, add and subtract */
```

```
    cout << add(10, 2) << endl;
    cout << subtract(10, 2) << endl;

    return 0;
}
```

As you can see we are including our **header file** in here as well, so we can use the **functions** declared in it, **add** and **subtract**. If you were to remove the **#include "mathPrimer.h"** from here, you will see that you cannot compile the code, because the compiler would not know anything about those functions, and it will complain about "**Undeclared identifiers**".

An important thing to note about **header files** is to never include **variables** in them, unless they are constants. **Header files** should only be used for declarations. Also, you should never include the definition for a **function** in the **header file** because it will change the whole scope of having a **header file**, and it will make it hard to read.

Also, you should always split the parts of your code into separate **header** and **source files** grouped by a certain criteria or **functionality**, because when only needing a part of it, you do not need to include all of the declarations that reside in your program. As in our example, we have called our **header file mathPrime.h**, as it will only contain basic math **functions**. If we want to make some other helper **functions**, for example, for printing to the screen and retrieving user input, we should make another pair of **header/source files**, that we should only include when needed.

Data Types in OOP

Data type

In C++ programming, we store the variables in our computer's memory, but the computer has to know what kind of data we want to store in them. The amount of memory required to store a single number is not the same as required by a single letter or a large number. Further, interpretation of different data is different inside computer's memory.

The memory in computer system is organized in bits and bytes. A byte is the minimum amount of memory that we can manage in C++. A byte can store a relatively small amount of data: one single character or a small integer. In addition, the computer can manipulate more complex data types that come from grouping several bytes, such as long numbers or non-integer numbers.

Data types in C++ is used to define the type of data that identifiers accepts in programming and operators are used to perform a special task such as addition, multiplication, subtraction and division etc of two or more operands during programming. C++ supports a large number of data types. Data types can be categorized into three types which are described below;

Built-in/Simple Data Types

There are four types of built-in data types and let us discuss each of these and the range of values accepted by them one by one.

- **Integer data type (int):**
  An integer is an integral whole number without a decimal point. These numbers are used for counting. For example 46, 167, -223 are valid integers. Normally an integer can hold numbers from -32768 to 32767. The int data type can further categorized into short, long and unsigned int.
  The short int data type is used to store integer with a range of -32768 to 32767, However, if the need be a long integer (long int) can also be used to hold integers from -2,147,483,648 to 2,147,483,648. The unsigned int can have only positive integers and its range lies up to 65536.

- **Floating point data type (float):**
  A floating point number has a decimal point. Even if it has an integral value, it must include a decimal point at the end. These numbers are used for measuring quantities. Examples of valid floating point numbers are: 35.5, -66.3, and 49.07.
  A float type data can be used to hold numbers from 3.4*10-38 to 3.4*10+38 with six or seven digits of precision. However, for more precision a double precision type (double) can be used to hold numbers from 1.7*10-308 to 1.7*10+308 with about 15 digits of precision.

- **Void data type:**
  It is used for following purposes;

  - It specifies the return type of a function when the function is not returning any value.
  - It indicates an empty parameter list on a function when no arguments are passed.
  - A void pointer can be assigned a pointer value of any basic data type.

- **Char data type:**
  It is used to store character values in the identifier. Its size and range of values is given in table below;

Name	Description	Size*	Range
Char	Character or small integer	1 byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2 bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer	4 bytes	signed: -2147483648 to 2147483648 unsigned: 0 to 4294967295
long int (long)	Long Integer	4 bytes	signed: -2147483648 to 2147483648 unsigned: 0 to 4294967295

Bool	Boolean value. It can take one of two values: true or false	1 byte	true or false
float	Floating point number.	4 bytes	+/- 3.4e +/-38 (~7 digits)
double	Double precision floating point number.	8 bytes	+/- 1.7e +/-308 (~15 digits)
long double	Long double precision floating point number	8 bytes	+/- 1.7e +/-308 (~15 digits)
wchar_t	Wide character	2 or 4 bytes	1 wide character

Derive Data Types

C++ also permits four types of derived data types. As the name suggests, derived data types are basically derived from the built-in data types. There are four derived data types. These are:

- Array
- Function
- Pointer and
- Reference

Array An array is a set of elements of the same data type that are referred to by the same name. All the elements in an array are stored at contiguous (one after another) memory locations and each element is accessed by a unique index or subscript value. The subscript value indicates the position of an element in an array.

Function A function is a self-contained program segment that carries out a specific well-defined task. In C++, every program contains one or more functions which can be invoked from other parts of a program, if required.

Reference A reference is an alternative name for a variable. That is, a reference is an alias for a variable in a program. A variable and its reference can be used interchangeably in a program as both refer to the same memory location. Hence, changes made to any of them (say, a variable) are reflected in the other (on a reference).

Pointer A pointer is a variable that can store the memory address of another variable. Pointers allow to use the memory dynamically. That is, with the help of pointers, memory can be allocated or de-allocated to the variables at run-time, thus, making a program more efficient

User Defined Data Types

C++ also permits four types of user defined data types. As the name suggests, user defined data types are defined are defined by the programmers during the coding of software development. There are four user defined data types. These are:

- Structure
- Union
- Class, and

- Enumerator

A structure allows for the storage, in contiguous areas of memory, of associated data items. A structure is a template for a new data type whose format is defined by the programmer. A structure is one of a group of language constructs that allow the programmer to compose his/her own data types.

```
struct [tag]
{
     [variable];
     members;
}
Example
struct Person
{
     char ssn[12]
          ,last_name[20]
          ,first_name[16]
          ,street[20]
          ,city[20]
          ,state[3]
          ,zip_code[11]
          ;
     int age;
     float height
          ,weight
          ;
     double salary;
};
```


Where **tag** is optional and only needs to be present if no **variable** is present. The **members** are variables declared as any C supported data type or composed data type. A structure is a set of values that can be referenced collectively through a variable name. The components of a structure are referred to as members of a structure. A structure differs from an array in that members can be of different data types. A structure is defined by creating a template. A structure template does not occupy any memory space and does not have an address, it is simply a description of a new data type. The name of the structure is the **tag**. The **tag** is optional when a **variable** is present and the **variable** is optional when the **tag** is present. Each member-declaration has the form

A Class specified by the keyword *class*, is a user defined type that contains both data members and member functions...

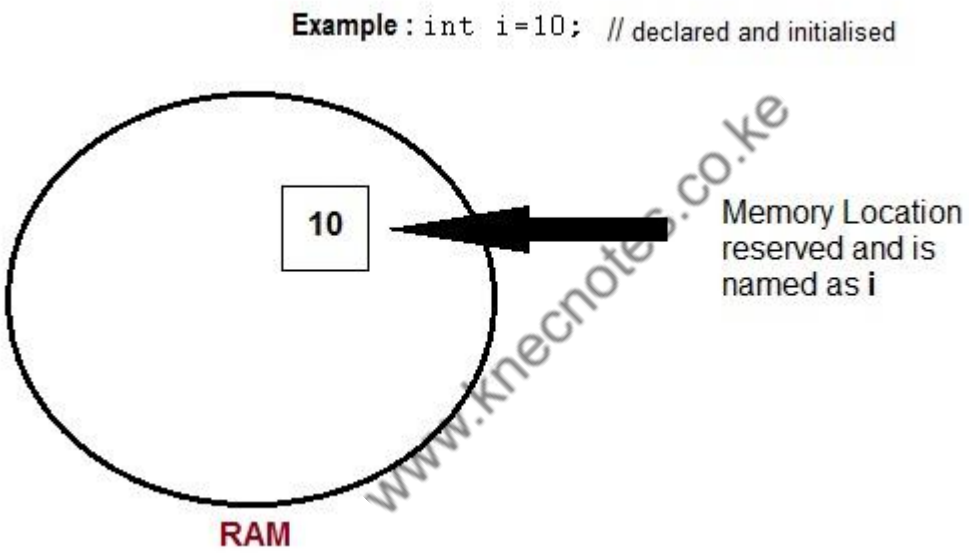
A union is a user-defined data or class type that, at any given time, contains only one object from its list of members (although that object can be an array or a class type).

An enumeration, specified by the keyword *enum*, is a set of integer constants associated by identifiers--called enumerators. Enumerations provide a manner to implement names (or identifiers), in place of **integer constants**. Enumerator values

begin at zero (0), if a value for the initial enumerator was not provided. Enumerators may be used wherever an int value is utilized. If no user specified value is assigned, compilers will assign the following integer value after the integer value assigned to the preceding enumerator.

Variable and variable declaration

Variables are named memory storage reserved for our programs to use (store, manipulate).
Variable are used in C++, where we need storage for any value, which will change in program. Variable can be declared in multiple ways each with different memory requirements and functioning. Variable is the name of memory location allocated by the compiler depending upon the data type of the variable.



Declaration and Initialization

Variable must be declared before they are used. Usually it is preferred to declare them at the starting of the program, but in C++ they can be declared in the middle of program too, but must be done before using them.

Example :

```
int i;      // declared but not initialized
char c;
int i, j, k;  // Multiple declaration
```

Initialization means assigning value to an already declared variable,

```
int i;   // declaration
i = 10;  // initialization
```

Initialization and declaration can be done in one single step also,

```
int i=10;         //initialization and declaration in same step
int i=10, j=11;
```

If a variable is declared and not initialized by default it will hold a garbage value. Also, if a variable is once declared and if try to declare it again, we will get a compile time error.

```
int i,j;
i=10;
j=20;
int j=i+j;   //compile time error, cannot redeclare a variable in same
scope
```

Program Example

Let's look at an example of how to declare an integer variable in the C++ language and use it.

a) Below is an example C++ program where we declare an integer variable and assign value in it:

```
#include <iostream>

int main()
{
   int age;

   age = 10;
   cout<<"The data in variable age is %d yrs.\n"<< age;

   return 0;
}
```

This C program would print "The `data in variable age is` 10 yrs."

b) Below is an example C++ program where we declare an integer variable and assign value in it throng an input statement "Cin":

```
#include <iostream>

int main()
{
   String name;
   int age;
   cout<<"Enter your name plz:\n";
   cin>>name;
   cout<<"Enter your age plz:\n";
   cin>>age;

   cout<<"Your Name is"<<name<<"and your age is"<<age<<"yrs"<< endl;

   return 0;
}
```

This C program would print `"Your Name is KIM and your age is 10 yrs".`

Scope of Variables

All the variables have their area of functioning, and out of that boundary they don't hold their value, this boundary is called scope of the variable. For most of the cases its

between the curly braces,in which variable is declared that a variable exists, not outside it. We will study the storage classes later, but as of now, we can broadly divide variables into two main types,

- Global Variables
- Local variables

Global variables

Global variables are those, which ar once declared and can be used throughout the lifetime of the program by any class or any function. They must be declared outside the `main()` function. If only declared, they can be assigned different values at different time in program lifetime. But even if they are declared and initialized at the same time outside the main() function, then also they can be assigned any value at any point in the program.

Example : Only declared, not initialized

```
include <iostream>
using namespace std;
int x;                 // Global variable declared
int main()
{
 x=10;                  // Initialized once
 cout <<"first value of x = "<< x;
 x=20;                  // Initialized again
 cout <<"Initialized again with value = "<< x;
}
```


Local Variables

Local variables are the variables which exist only between the curly braces, in which its declared. Outside that they are unavailable and leads to compile time error.

Example :

```
include <iostream>
using namespace std;
int main()
{
 int i=10;
 if(i<20)         // if condition scope starts
  {
    int n=100;    // Local variable declared and initialized
  }               // if condition scope ends
 cout << n;       // Compile time error, n not available here
}
```

Some special types of variable

There are also some special keywords, to impart unique characteristics to the variables in the program. Following two are mostly used, we will discuss them in details later.

1. **Final** - Once initialized, its value cant be changed.
2. **Static** - These variables holds their value between function calls.

Example :

```
include <iostream>
using namespace std;
int main()
{
 final int i=10;
 static int y=20;
{
```

Type Conversion and Type Casting

In computer science, **type conversion** or **typecasting** refers to changing an entity of one datatype into another. There are two types of conversion: implicit and explicit. The term for implicit type conversion is **coercion**. Explicit type conversion in some specific way is known as **casting**. Explicit type conversion can also be achieved with separately defined conversion routines such as an overloaded object constructor.

Both Type conversion and Type casting in C++ are used to convert one predefined type to another type.
Type Conversion is the process of converting one predefined type into another type.
and type Casting is the converting one predefined type into another type forcefully.

Need of Type Conversion and Type Casting in C++

An Expression is composed of one or more operations and operands. Operands consists of constants and variables. Constants and expressions of different types are mixed together in an expression. so they are converted to same type or says that a conversion is necessary to convert different types into same type.

Types of Type Conversions

C++ facilitates type conversion into 2 forms :

- Implicit Type Conversion
- Explicit Type Conversion

Implicit Type Conversions :

Implicit Type Conversion is the conversion performed by the compiler without programmer's intervention.
It is applied, whenever, different data types are intermixed in an expression, so as not to loose information.

The C++ compiler converts all operands upto the type of the largest operand, which is called **type promotion.**

Usual Arithmetic Conversions are summarized in the following table –

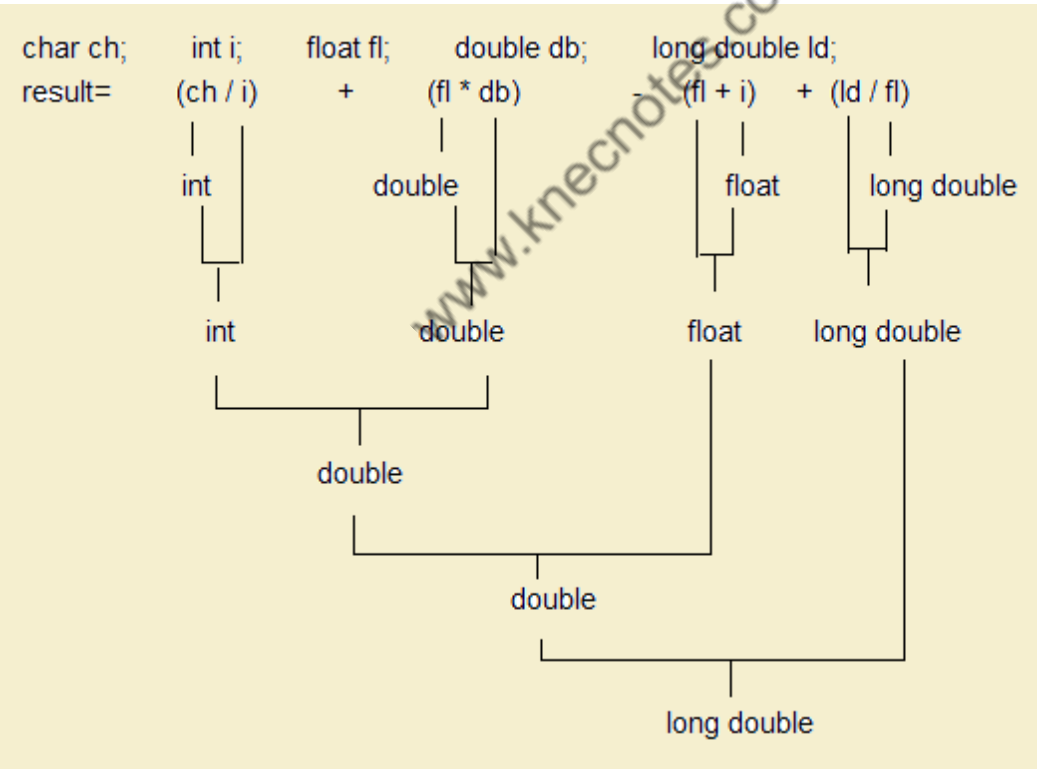
StepNo.	If either'stype of	Then resultant type of other operand	Otherwise

1	long double	long double	Step 2
2	double	double	Step 3
3	float	float	Step 4
4	—	integral promotion takes place followed by step 5	—
5	unsigned long	unsigned long	Step 6
6	long int ant the other is unsigned int	(i) long int (provided long int can represent all values of unsigned int)	Step 7
		(ii) unsigned long int(if all values of unsigned int can't be represented by long int)	Step 7
7	long	long	Step 8
8	unsigned	unsigned	Both operandsare int

The step 1 and 2 in the above table will be read as –
Step 1: If either operand is of type **long double**, the other is converted to **long double.**
Step2 : Otherwise, if either is of type **double**, the other is converted to **double.**

After applying above arithmetic conversions, each pair f operands is of same type and the result of each operation is the same as the type of both operands.

Example of Implicit Type Conversion :



Explicit Type Conversion :

Explicit Type conversion is also called type casting. It is the conversion of one operand to a specific type. An explicit conversion is a user defined that forces an expression to be of specific type.

Syntax : (type) expression

Example : float(a+b/5) ; This expression evaluates to type float.

Problem in Explicit Type Conversion :

Assigning a value of smaller data type to a larger data type, may not pose any problem. But, assigning a value of larger data type to smaller type, may poses problems. The problem is that assigning to a smaller data type may loose information, or result in losing some precision.

Conversion Problems –

S.no	Conversion	Potential Problems
1	Double to float	Loss of precision(significant figures)
2	Float to int	Loss of fractional part
3	Long to int/short	Loss of Information as original valuemay be out of range for target type

Type Compatibility

In an assignment statement, the types of right types and left side of an assignment should be compatible, so that conversion can take place. For example,

ch=x; (where ch is of char data type and x is of integer data type)

How and Why Information is loose ?

what is Big Endian ?? ⇒ refer to the link Click here

since the memory representation in ***Big-Endian,*** Let

int x=1417;
ch=x;

now, x will be 00000101 10001001 in binary.