UNIT-4

Dr. Shipra Saraswat
UNIT-4 Notes
OOSD

C++ Basics :-

The basic structure of a C program is divided into 6 parts which makes it easy to read, modify, document, and understand in a particular format.

Sections of the C program :-

1) Documentation
2) Preprocessor Section
3) Definition
4) Global Declaration
5) Main () Function
6) Sub Programs

1) Documentation

This Section Consists of the description of the program, the name of the program, and the creation date and time of the program. It is specified at the staut of the program in the form of Comments. Documentation Can be represented as :-

```
// description, name of the program, programmer
   name, date, time etc.
```

2) Preprocessor Section

All the header files of the program will be declared in the preprocessor section of the program. Header files help us to access other's improved code into our code.

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

3. Definition

Preprocessors are the programs that process our source code before the process of compilation. There are multiple steps which are involved in the writing and execution of the program. Preprocessor directives start with the '#' symbol. The # define preprocessor is used to create a constant throughout the program.

```
# define long long ll
```

4. Global Declaration

The global declaration section contains global variables, function declaration, and static variables. Variables and functions which are declared in this scope can be used anywhere in the program.

```
int num = 18;
```

5. main() Function

Every C program must have a main function. The main() function of the program is written in this section. operations like declaration and execution are performed inside the curly braces of the main program. The return type of the main() function can be int as well as void too. Void() main tells the Compiler that the program will not return any value. The int main() tells the Compiler that the program will return an integer value.

```
Void main ()
```

6. Sub programs

User defined functions are called in this section, of the program. The control of the program is shifted to the called function whenever they are called from the main or outside the main() function. These are specified as per the requirements of the programmer.

```
int Sum(int x, int y)
{
  return x+y;
}
```

Namespace

A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it. Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries.

Identifiers

The C++ identifier is a name used to identify a variable, function, class, module or any other user defined item. An identifier starts with a letter A to Z or a to z, an underscore '_' followed by zero or more letters, underscores, and digits (0 to 9). C++ does not allow punctuation characters such as @, $, and % within identifiers. C++ is a case sensitive programming language.

Variables

Variables are containers for storing data values. In C++, there are different types of variables (defined with different keywords), for example

- → int stores integers (whole numbers)
- → double stores floating point number such as 19.8
- → char stores single characters, such as 'a' or 'B'. char values are surrounded by single quotes.
- → string stores text, such as "Hello world". String values are surrounded by double quotes
- → bool Stores values with two states : true or false.

Constants

Constants refer to fixed values that the program may not alter and they are called literals. Constants can be of any of the basic data types and can be divided into integer numerals, floating point numbers, characters, strings and boolean values. In c/c++ program we can define Constants in two ways as shown below :-

1) Using # define preprocessor directive
2) Using a Const Keyword

Literals :- The values assigned to each Constant variables are referred to as the literals.

1) Using #define preprocessor directive :- This directive is used to declare an alias name for existing variable or any value.

```
#define identifierName value
```

identifierName :- It is the name given to constant.
value :- This refer to any value assigned to identifier Name.

2) Using a Const keyword :- Using Const keyword to define constants is as simple as defining variables, the difference representation is :-

```
Const int var;  X
Const int var;
      var = 5;  X

Const int var = 5;
```

Below program shows how to use Const to declare Constants of different data types :-

```
#include <stdio.h>
Const
{
  // int Constant
  Const int intVal = 10;

  // Real Constant
  Const float floatVal = 4.14;

  // char Constant
  Const char charVal = 'A';

  // String Constant
  Const char StringVal [10] = "ABC";
```

```
Printf ("Integer Constant : %d \n", intVal );
Printf ("Floating Point Constant : %.2f\n", floatVal);
Printf ("character Constant : %c\n", charVal );
Printf ("String Constant : %s\n", StringVal );

 return 0;
}
```

Output :-

```
Integer Constant : 10
Floating Point Constant : 4.14
character Constant : A
String Constant : ABC
```

Enum (Enumeration)

Enum or enumeration is a data type consisting of named values like elements, members etc, that represent integral constants. It provides a means to declare and arrange necessary constants. It also makes the code easy to maintain and less complex.

→ enum is a user defined data type which can be assigned some limited values. These values are defined by the programmer at the time of declaring the enumerated type.

```
enum enumerated-type-name { value1, value2,
                            value3 ..... valueN};
```

Operators in C++

An operator is a symbol that operates on a value to perform specific mathematical or logical computations. Following are the operators in C++ :-

1) Arithmetic operators
2) Relational operators
3) Logical operators
4) Bitwise operators
5) Assignment operators
6) Ternary or Conditional operators.

Arithmetic operators

1) Unary operators (++) and (--)
   Increment operator, Decrement operator

2) Binary operators (+) and (-), *, /, %

Relational operators

==, >, >=, <, <=, !=

== is equal to
> greater than
>= greater than equal to
< less than
<= less than or equal to
!= Not equal to

Logical operators

1) Logical AND &&
2) Logical OR ||
3) Logical NOT !

Bitwise operators

1) Binary AND &
2) Binary OR |
3) Binary XOR ^
4) Left shift <<
5) Right shift >>
6) one's Complement ~

Assignment operators

1) Assignment operator =
2) Add and Assignment operator +=
3) Subtract and assignment operator -=
4) multiply and assignment operator *=
5) Divide and assignment operator /=

Ternary and Conditional operators

Expression 1 ? Expression 2 : Expression 3

There are some other Common operators available in C++ besides the above mentioned operators

1) Sizeof Operator
2) Comma operator
3) -> operator → used to access the variables of classes or structures.
4) Cast operator → used to Convert one data type into another.
5) Dot operator → used to accesses members of structure variables or class objects in C++.
6) & operator → used to represent the memory address.
7) * operator → Indirection operator

Type Casting

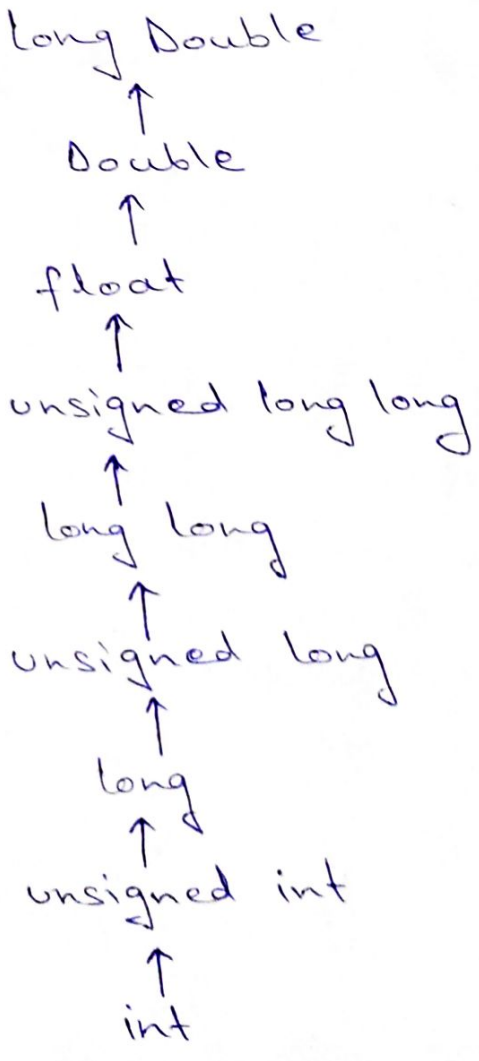
Type casting is the process of converting one data type to another data type by the programmer using the casting operator during program design.

```
int x;
float y;
y=(float)x;
```

Two types of type casting in C

1) Implicit Type Casting

It is used to convert the data type of any variable without using the actual value that the variable holds. It performs the conversions without altering any of the values which are stored in the data variable. Conversion of lower data type to higher data type will occur automatically.



2) Explicit Type Casting

There are some cases where if the data.type remains unchanged, it can give incorrect output. In such cases, typecasting can help to get the correct output and reduce the time of compilation. In explicit type casting, we have to force the conversion between data types. This type of casting is explicitly defined within the program.

Advantages of Type Casting

1) Type casting in C programming makes the program very lightweight.
2) Type representation and hierarchies are some features we can take advantage of with the help of typecasting.
3) Type casting helps programmers to convert one data type to another data type.

Control Structures

Control structures are just a way to specify flow of control in programs. Any algorithm or program can be more clear and understood if they use self contained modules called as logic or control structures. It basically analyzes and chooses in which direction a program flows based on certain parameters or conditions. There are three basic types of logic, or flow of control.

1) Sequence logic or Sequential flow
2) Selection logic or Conditional flow
3) Iteration logic or repetitive flow

Inline Functions

Inline function is a function that is expanded in line when it is called. when the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call.

Syntax :-

```
inline return-type function-name (parameters)
{
 // function code
}
```

Macro Functions

It is also called preprocessor directive. The macros are defined by the # define keyword. Before the program Compilation, the preprocessor examines the program whenever the preprocessor detects the macros then preprocessor replaces the macro by the macro definition.

```
# define MACRO_NAME Macro-definition
```

Difference between Inline and macros in C++

Inline	Macro
1) An inline function is defined by the inline keyword.	1) whereas the macros are defined by the # define keyword.
2) Through inline function the class data members can be accessed.	2) whereas macro can't access the class's data members.

3) In the case of inline function, the program can be easily debugged.	3) whereas in the case of macros, the program can't be easily debugged.
4) In the case of inline, the arguments are evaluated only once.	4) whereas in the case of macro, the arguments are evaluated every time whenever macro is used in the program.
5) In C++, inline may be defined either inside the class or outside the class.	5) whereas the macro is all the time defined at the beginning of the program.
6) In C++, inside the class, the short length functions are automatically made the inline functions.	6) while the macros is specifically defined.
7) Inline is not as widely used as macros.	7) while the macro is widely used.
8) Inline is not used in competitive programming.	8) while the macro is very much used in competitive programming.
9) Inline function is terminated by the curly brace at the end.	9) while the macro is not terminated by any symbol, it is terminated by a new line.

Default Arguments

A default argument is a value provided in a function declaration that is automatically assigned by the Compiler if the calling function does not provide a value for the argument. In case any value is passed, the default value is overridden.

```
// CPP Program to demonstrate Default Arguments
#include <iostream>
using namespace std;

// A function with default arguments,
// it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int Sum(int x, int y, int z = 0, int w = 0) // assigning default

{ return (x + y + z + w);
}
// Driver Code
int main()
{ // statement 1
  Cout << sum(10, 15) << endl;
  // statement 2
  Cout << sum(10, 15, 25) << endl;
// statement 3
Cout << sum(10, 15, 25, 30) << endl;
  return 0;
}
```

output

```
25
50
80
```

Friend Function

A friend function of a class is defined outside that class's scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend can be a function, function template, or member function, or a class or class template in which case the entire class and all of its members are friends.

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword friend as follows :-

Consider the following program

```
# include <iostream>
using namespace std;
class Box {
   double width;

public:
    friend void printwidth (Box box);
    void setwidth (double wid);
  };
// member function definition
void Box :: setwidth (double wid) {
  width = wid;
}

// Note: printwidth() is not a member function
  of any class
```

```
void printwidth (Box box) {
/* Because printwidth() is a friend of Box, it
 can directly access any member of this class */
cout << "width of box : " << box.width << endl;
 }
// main function for the program
int main() {
 Box box;

// set box width without member function
 box.setwidth (10.0);

// use friend function to print the width
 printwidth (box);
 return 0;
}
```

output:- width of box : 10

Virtual Function

A virtual function is a member function which is declared within a base class and is re-defined (overridden) by a derived class. when you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

→ virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.

→ They are mainly used to achieve run time Polymorphism.

→ Functions are declared with a virtual keyword in base class.

→ The resolving of function call is done at runtime.

Rules for virtual Functions

1) virtual functions cannot be static.
2) A virtual function can be a friend function of another class.
3) virtual functions should be accessed using pointer or reference of base class type to achieve runtime polymorphism.
4) The prototype of virtual functions should be the same in the base as well as derived class.
5) They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re define the virtual function) in that case, the base class version of the function is used.
6) A class may have virtual destructor but it cannot have a virtual constructor.

// Program to illustrate concept of virtual Functions

```
#include <iostream>
using namespace std;
class base {
public:
    virtual void print()
    {
    cout << "print base class \n";
    }
    void show()
    {
    cout << "show base class \n";
    }
};
class derived : public base {
public:
void print()
{
  cout << "print derived class \n";
}
 void show()
{
  cout << "show derived class \n";
  }
};
int main()
{
    base *bptr;
    derived d;
    bptr = &d;
```

// virtual function, binded at runtime

```
  bptr->print();
```

```
// non virtual function, binded at compile
time
  bptr->show();
return 0;
}
```

output

```
Print derived class
Show base class
```