VIT
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

BCSE102L- Structured and object-oriented programming

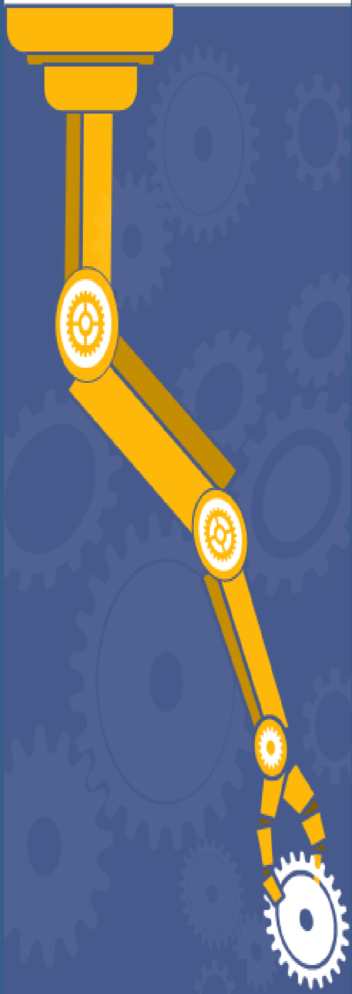
Dr. P.Keerthika
Associate Professor
School of Computer Science & Engineering
VIT, Vellore.

BCSE102L- Structured and object-oriented programming

Module	Title	Hours
Module:1	**C Programming Fundamentals**	**2 hours**
Variables - Reserved words – Data Types – Operators – Operator Precedence - Expressions - Type Conversions - I/O statements - Branching and Looping: if, if-else, nested if, if-else ladder, switch statement, goto statement - Loops: for, while and do…while – break and continue statements.		
Module:2	**Arrays and Functions**	**4 hours**
Arrays: One Dimensional array - Two-Dimensional Array – Strings and its operations. User Defined Functions: Declaration – Definition – call by value and call by reference - Types of Functions - Recursive functions - Storage Classes - Scope, Visibility and Lifetime of Variables.		
Module:3	**Pointers**	**4 hours**
Declaration and Access of Pointer Variables, Pointer arithmetic – Dynamic memory allocation – Pointers and arrays - Pointers and functions.		
Module:4	**Structure and Union**	**2 hours**
Declaration, Initialization, Access of Structure Variables - Arrays of Structure - Arrays within Structure - Structure within Structures - Structures and Functions – Pointers to Structure -		
Module:5	**Overview of Object-Oriented Programming**	**5 hours**
Features of OOP - Classes and Objects - "this" pointer - Constructors and Destructors - Static Data Members, Static Member Functions and Objects - Inline Functions – Call by reference - Functions with default Arguments - Functions with Objects as Arguments - Friend Functions and Friend Classes.		
Module:6	**Inheritance**	**5 hours**
Inheritance - Types of Inheritance: Single inheritance, Multiple Inheritance, Multi-level Inheritance, Hierarchical Inheritance - Multipath Inheritance - Inheritance and constructors.		
Module:7	**Polymorphism**	**4 hours**
Function Overloading - Operator Overloading – Dynamic Polymorphism - Virtual Functions - Pure virtual Functions - Abstract Classes.		
Module:8	**Generic Programming**	**4 hours**
Function templates and class templates, Standard Template Library.		
	Total Lecture hours:	**30 hours**

BCSE102L- Structured and object-oriented programming – Text Books and Reference Books

Text Book(s)	
1.	Herbert Schildt, C: The Complete Reference, 4th Edition, McGraw Hill Education, 2017
2.	Herbert Schildt, C++: The Complete Reference, 4th Edition, McGraw Hill Education, 2017.
Reference Books	
1.	Yashavant Kanetkar, Let Us C: 17th Edition, BPB Publicaitons, 2020.
2.	Stanley Lippman and Josee Lajoie, C++ Primer, 5th Edition, Addison-Wesley publishers, 2012.



BCSE102P- Structured and object-oriented programming Laboratory

Indicative Experiments	
1.	Programs using basic control structures, branching and looping
2.	Experiment the use of 1-D, 2-D arrays and strings and Functions
3.	Demonstrate the application of pointers
4.	Experiment structures and unions
5.	Programs on basic Object-Oriented Programming constructs.
6.	Demonstrate various categories of inheritance
7.	Program to apply kinds of polymorphism.
8.	Develop generic templates and Standard Template Libraries.

Text Book(s)	
1.	Robert C. Seacord, Effective C: An Introduction to Professional C Programming, 1st Edition, No Starch Press, 2020.
Reference Book(s)	
1.	Vardan Grigoryan and Shunguang Wu, Expert C++: Become a proficient programmer by learning coding best practices with C++17 and C++20's latest features, 1st Edition, Packt Publishing Limited, 2020.

BCSE102L- Structured and Object-Oriented Programming

- **Module-5: Overview of Object Oriented Programming**
  - **Features of OOP- Classes and Objects**
  - **Constructors and Destructors**
  - **Static Data Members and Member Functions**
  - **Inline Functions, Call by reference**
  - **Functions with Default Arguments**
  - **Functions with Objects as Arguments**
  - **Friend Functions and classes**

VIT
Vellore Institute of Technology

Functions- Recall

- Functions play an important role in C Program development.
- Dividing a program into functions is one of the major principles of top-down, structured programming.
- Also it is possible to reduce the size of a program by calling and using them at different places in the program.

```
void show();      /* Function declaration */
main()
{
     .....
     show();          /* Function call */
     .....
}
void show()       /* Function definition */
{
     .....
      .....       /* Function body */
      .....
  }
```

Functions- Definition and Declaration

```
Type name_of_the_function (argument list)
      {
        //body of the function
      }


   void add(int a, int b) //variable names are must in definition
   {
     int sum;
      sum=a+b;
      cout<<"\nThe sum of two numbers is "<<sum<<endl;
      }
```

Arguments to a Functions-Example

```
void main( )
{
    int sqr(int);
    int num, s;

    clrscr( );
    cout<<"Enter the number \n";
    cin>>num;

    s=sqr(num);        → num- Actual parameter

    cout<<"num="<<num<<endl;
    cout<<"square of number="<<s<<endl;
}
    int sqr(int x)     → x- Formal parameter
    {
        int t;
        t=x*x;
        return t;
    }
```

Parameter Passing to Functions

The parameters passed to function are called **actual parameters.**

The parameters received by function are called **formal parameters.**

- **Pass/Call by Value:** In this parameter passing method, values of actual parameters are copied to function's formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of caller.
- **Pass/Call by Reference:** Both actual and formal parameters refer to same locations, so any changes made inside the function are actually reflected in actual parameters of caller.*(Definition based on Pointers).*
  - Passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. *(Definition based on Reference variable – in C++)*

Pass/Call by Value – C program

```
int main()
{
        int m = 6, n = 10;
        printf("Before swapping value of m and n is :%d,%d\n",m,n);
        printf("Before swapping address of m and n is :%u,%u\n",&m,&n);
        swap(m,n);
        printf("After swapping m,n value is : %d,%d\n",m,n);
        printf("After swapping (m,n) address is :%u,%u\n",&m,&n);
        return;
}
int swap(int x, int y)
{
        int tmp = x;
        x = y;
        y = tmp;
        printf("Swapped values (x,y) is : %d,%d\n",x,y);
        printf("Swapped values address (x,y) is : %u,%u\n",&x,&y);
}
```

```
Before swapping value of m and n is :6,10
Before swapping address of m and n is :6422044,6422040
Swapped values (x,y) is : 10,6
Swapped values address (x,y) is : 6422000,6422008
After swapping m,n value is : 6,10
After swapping (m,n) address is :6422044,6422040
```

Pass/Call by Value – C++ program

```
#include <iostream>
using namespace std;
void swap(int x, int y);
int main ()
{
        int a = 100; int b = 200;
        cout << "Before swap, value of a :" << a << endl;
        cout << "Before swap, value of b :" << b << endl;
        swap(a, b); // calling a function to swap the values.
        return 0;
}
void swap(int x, int y)
{
        int temp=x;
        x=y;
        y=temp;
        cout << "After swap, value of a :" << x << endl;
        cout << "After swap, value of b :" << y << endl;
}
```

Pass/Call by Address- C++ Program

```
void swap(int *x, int *y);
int main ()
{
    int a = 100, b = 200;
    cout << "Before swap, value of a :" << a << endl;
    cout<< "Before swap,  address of a :" << &a << endl;
    cout << "Before swap, value of b :" << b << endl;
    cout<< "Before swap,  address of b :" << &b << endl;
    swap(&a, &b);
    cout << "After swap, value of a :" << a << endl; //Value of a printed
    cout << "After swap, value of b :" << b << endl;// Value of b printed
}
int swap(int *x, int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
    cout << "After swap, value of x :" << x << endl; //Value of x printed
    cout << "After swap, value of y :" << y << endl;// Value of y printed
}
```

```
Before swap, value of a : 100
Before swap, address of a : 6422044
Before swap, value of b : 200
Before swap, address of b : 6422040
After swap, value of x : 6422044
After swap, value of y : 6422040
After swap, value of a : 200
After swap, value of b : 100
```

Pass/Call by Reference- C++ Program

```
void swap(int &x, int &y);
int main ()
{
    int a = 100, b = 200;
    cout << "Before swap, value of a :" << a << endl;
    cout<< "Before swap,  address of a :" << &a << endl;
    cout << "Before swap, value of b :" << b << endl;
    cout<< "Before swap,  address of b :" << &b << endl;
    swap(a, b);
    cout << "After swap, value of a :" << a << endl; //Value of a printed
    cout << "After swap, value of b :" << b << endl;// Value of b printed
}
int swap(int &x, int &y)    ---- x and y are called reference variables
{
    int temp= x;    x = y;
    y = temp;
    cout << "After swap, value of x :" << x << endl; //Value of x printed
    cout<<"Address of x:"<<&x<<endl;
    cout << "After swap, value of y :" << y << endl;// Value of y printed
    cout<<"Address of y:"<<&y<<endl;
}
```

```
Before swap, value of a :100
Before swap,  address of a :0x61fe1c
Before swap, value of b :200
Before swap,  address of b :0x61fe18
After swap, value of x :200
Address of x:0x61fe1c
After swap, value of y :100
Address of y:0x61fe18
After swap, value of a :200
After swap, value of b :100
```

- *Pass parameters to the function by reference*
- *Formal arguments become alias to actual arguments*
- *Here exchange the values of a and b using their alias(reference variables) x and y*

Difference between CBA &CBR

Call by Address	Call by reference

void change(int *, int *);
void change(int * x, int *y)
{
*x = *x + 10;
*y = *y + 20;
}
Main()
{
int a = 10, b = 20;
change (&a, &b);
}
	void change(int &, int &);
	void change(int &x, int &y)
	{
	x = x + 10;
	y = y + 20;
	}
	Main()
	{
	int a = 10, b = 20;
	change (a, b);
	}

Address of a and b are passed and new location are created for formal parameters x and y	Only reference of a and b is created i.e., new name for the location a and b is created, but no new location is created for x and y.

References vs Pointers

- **References are often confused with pointers but three major differences between references and pointers are −**

- You cannot have NULL references. You must always be able to assume that a reference is connected to a piece of storage.
- Once a reference is initialized to an object, it cannot be changed to refer to another object. Pointers can be pointed to another object at any time.
- A reference must be initialized when it is created. Pointers can be initialized at any time.

Inline Functions- Why??

- Main objective of functions in a program is to save memory.
- Every time when function is called it takes lot of time to execute series of instructions for tasks(For ex: jumping to function, saving registers, push arguments and return to the calling function).
- C having solution – MACROS – simply called as pre-processor macros.
- But they are not functions, and also error checking does not occur.
- C++ - To eliminate the cost of calls to small functions- it proposes **inline function**.(Similar to Macros)
- Termed as macros expansion.

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.
- It is a method to optimize the code and enhance a program's performance.
- It improves the execution time and speed of the program.

SYNTAX

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

Inline Functions

```
#include <iostream.h>
using namespace std;
inline int max(int x,int y)
{
        return x>y ? x :y;
}
int main( )
{
        int n1,n2;
        cout<<"Enter two numbers \n";
        cin>>n1>>n2;
        int m=max(n1,n2);
        cout<<"Max="<<m<<endl;
        return 0;
}
```

Square of Number using Inline Functions

```
#include <iostream.h>
inline int square (int x)
{
        return x*x;
}
int main( )
{
        int s1=square (2+3);
        int s2=square (3*4-5+6/2);
        int s3=square (++s1);
        cout<<"s1="<<s1<<endl;
        cout<<"s2="<<s2<<endl;
        cout<<"s3="<<s3<<endl;
        return;
}
```

*Before expanding the function as function call is encountered compiler first computes the expression written as argument and make a single argument. So square (2+3) first turned into square (5) then expanded, similarly square (3*4-5+6/2) turned into square (10) and finally square (++s1) turned into square (26).*

Square of Number using Macro Functions

```
#include<iostream.h>
#define SQR(x) (x*x)
int main( )
{
        int s1=SQR(2+3);
        int s2=SQR(3*4−5+6/2);
        int s3=SQR(++s1);
        cout<<“s1=”<<s1<<endl;
        cout<<“s2=”<<s2<<endl;
        cout<<“s3=”<<s3<<endl;
        return 0;
}
```

When macro SQR is expanded all three statements appear as :

*s1=2+3*2+3;*

*s2=(3*4−5+6/2)*(3*4−5+6/2)*

*s3= ++s1 * ++s1*

Some points in Inline Functions

- Inline function executes faster than normal function.
- Inline function does not work as inline when code contains loops, recursions, goto, switch, static variable etc.
- Inline function does not work if a function contains
  - static variables
  - Recursive call (Recursive functions)
  - that contain switch or go-to statements
  - that contain while, do-while, or for looping controls
  - with a variable number of arguments (Var_Args)
  - function definition is too big or complicated

Functions with Default arguments

- **In C ++ is it possible for a function not to specify all its arguments??**
- **YES**
- Some of the arguments may be specified their default values at the time of declaring the function.
- When a function having default argument is called, compiler checks for the number and type of argument as well as was not specified during the function call, default value of that argument is assumed.
- For Example:

void show (int x, int y = 20); // *two argument of int type out of which second argument is default.*

1. *show(10) – x is assigned 10 and y is assigned 20 as default*
2. *show(10,100) – x is assigned 10 and y is assigned 100 because 20 is overridden.*

Points to remember when using default arguments

- In a function with default argument, if one argument is default, all successive arguments must be default.
- We cannot provide default values in the middle of the arguments or towards left side.
- Examples:

```
void show (int x, int y = 20, int z=35); // legal
void show (int x, int y = 30, int z); // Illegal
void show (int x=20, int y); // Illegal
void show (int x=10, int y = 20, int z=35); // legal
```

Default arguments

```
#include <iostream.h>
int main( )
{
        void show(char* s=”Good Morning”);
        show( );
        show(“Good Evening”);
        return 0;
}
void show(char *s)
{
        cout<<“Argument to show :”<<s<<endl;
}
```

Argument to show : Good Morning
Argument to show : Good Evening

Default arguments

```
int main( )
{
int incr(float &sal, float bonus_pr=10);
        float s;
        cout<<"Enter the salary:"<<endl;
        cin>>s;
        if(s>=5000)
                incr(s,15);
        else
                incr(s);
        cout<<"Salary with bonus="<<s<<endl;
        return 0 ;
}
int incr(float & sal, float bonus_pr)
{
sal = sal*(1+bonus_pr/100);
}
```

Enter the salary : 7000
Salary with bonus=8050

Functions with Objects as arguments

- In C++, Similar to returning and passing arguments to function of basic type like int, char, double, float, char* etc.
- ***we can pass objects of class to functions and even return objects from functions(Passing and returning object)***

demo show(demo);

// For a function show which takes an object of demo class type and return an object of demo class.

Example- passing objects to function

```
class demo
{
int num;
public :
void input(int x)
        {
                num=x;
        }
void copy(demo);
void show( )
{
cout<<"num="<<num<<endl;
}
};
```

```
void demo : :copy (demo d)
{
num=d.num;
}
void main( )
{
demo d1,d2;
d1.input(20);
d2.copy(d1);
cout<<"Object d1\n"<<endl;
d1.show( );
cout<<"Object d2\n";
d2.show( );
}
```

OUTPUT:

Object d1
num=20
Object d2
num=20

Example- passing objects to function

```
class employee
{
float sal;
char name[10];
public :
void input(float s, char n[ ])
{
        sal=s;
        strcpy(name,n);
}
void comp_sal(employee temp);
};
```

OUTPUT:

Ravi salary is higher than Hari salary

```
void employee : : comp_sal(employee temp)
{
if(temp.sal>sal)
cout<<temp.name<<“ ‘ salary is higher “
<<“ than “ <<name<<“ ‘ salary”<<endl;
else
cout<<name<<“‘ salary is higher “
<<“ than “<<temp.name<<“ ‘salary\n”;
}
void main( )
{
employee e1, e2;
e1.input(10000.2,5”Hari”);
e2.input(11000.10,”Ravi”);
e1.comp_sal(e2);
getch( );
}
```

Example- Returning objects as arguments

```
class demo
{
int num;
public :
void input(int x)
{
num = x;
}
demo copy( );
void show( )
{
cout<<"num="<<num<<endl;
}
};
```

OUTPUT:
Object d1
num=20
Object d2
num=20

```
demo demo : :copy( )
{
demo temp;
temp.num = num;
return temp;
}
void main( )
{
        demo d1, d2;
        d1.input(20);
        d2=d1.copy( );
        cout<<"Object d1\n";
        d1.show( );
        cout<<"Object d2\n";
        d2.show( );
}
```

Practice Exercise

- Create a class time that has separate int member data for hours, minutes and seconds. One constructor should initialize this data to 0, and another should initialize it to fixed values. Another member function should display it, in 11:59:59 format. The final member function should add two objects of type time passed as arguments. A main program should create two initialized time objects (should they be const?) and one that is not initialized. Then it should add the two initialized values together, leaving the result in the third time variable. Finally, it should display the value of this third variable. Make appropriate member functions const.