Department of Computer Science and Engineering

PES UNIVERSITY

Object Oriented Programming with C++
UE22CS221A

UNIT 1

Ms. Kusuma K V

Introduction

C++ was designed by Bjarne Stroustrup at AT&T Bell Laboratories in the year 1980.

The initial version of C++ was called 'C with classes'. In 1983, "C with Classes" was renamed to "C++ ". In fact, C++ can be considered as a superset of C. C++ supports all the features of C in addition to providing new features of its own. C++ was developed to combine the procedural programming features of C and object-oriented features of Simula. Simula is the first language to support object-oriented programming features.

An international standard for C++ was developed in 1998 by the International Standards Organization (ISO). The C++ programming language was initially standardized in 1998, which was then amended by the C++03, C++11 and C++14, C++17 standards. C++20 is the standard being currently worked on. C++23 is the next standard, keeping with the current trend of a new version every three years. It is expected to be released in the month of December.

C++ used in :

- Games
- Graphic User Interface (GUI) based applications
- Web Browsers
- Advance Computations and Graphics
- Database Software
- Operating Systems
- Enterprise Software
- Medical and Engineering Applications
- ...

Features of C++, Object Oriented Concepts

- Data Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Object-oriented programming: Method of writing programs using data abstraction, inheritance, and dynamic binding.

Dynamic binding: Delaying until run time the selection of which function to run. In C++, dynamic binding refers to the runtime choice of which virtual function to run based on the underlying type of the object to which a reference or pointer is bound. (Discussed in Unit 4).

Data abstraction

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

A real life example: Consider a fan. We turn the fan on and off using ***switch***, regulate the speed using ***regulator*** but we do not know the internal details of how these operations are carried out. Thus, we can say a fan clearly separates its internal implementation from its external interface and you can play with its interfaces like switch, regulator without having any knowledge of its internals.

The interface of a class consists of the operations that users of the class can execute. The implementation includes the class' data members, the bodies of the functions that constitute the interface.

Encapsulation

Encapsulation is defined as binding together the data and the functions that manipulates them. Encapsulation enforces the separation of a class' interface and implementation. A class that is encapsulated hides its implementation—users of the class can use the interface but have no access to the implementation.

In C++ we use access specifiers to enforce encapsulation:

- Members defined after a **public** specifier are accessible to all parts of the program. The public members define the interface to the class.
- Members defined after a **private** specifier are accessible to the member functions of the class but are not accessible to code that uses the class. The private sections encapsulate (i.e., hide) the implementation.

A class that uses data abstraction and encapsulation defines an abstract datatype. In an abstract data type, the class designer worries about how the class is implemented. Programmers who use the class need not know how the type works. They can instead think abstractly about what the type does.

Inheritance

Inheritance is a programming feature that lets a type inherit the interface of another type. Through inheritance, we can define classes that model the relationships among similar types. When we define a class as publicly inherited from another, the derived class should reflect an "Is A" relationship to the base class.

Polymorphism

Polymorphism is derived from a Greek word meaning "many forms." In C++ polymorphism is mainly divided into two types:

- Compile time Polymorphism: This type of polymorphism is achieved by function overloading or operator overloading.
- Runtime Polymorphism: This type of polymorphism is achieved by Function Overriding.

Composition

Composition is "has a" relationship between objects.

Simple Input/ Output (IO) Operations

IO is provided by the standard library. iostream library handles formatted input and output.
Fundamental to iostream library are 2 types which represent input and output streams respectively:

- istream
- ostream

Note: Stream is a sequence of characters intended to be read from or written to IO device of some kind.

- cout is an object of ostream class. It stands as an alias for the console output device i.e., the monitor.
- Output operator (<<) originally left shift operator, has had its definition extended in C++. In the given context, it operates as the insertion operator also called put to operator.

- It is a binary operator. The operand on its left must be some object of ostream class. The operand on its right must be a value of some fundamental data type.
- The value on the RHS of the insertion operator is inserted into the stream headed towards the device associated with the object on its left.
- Because the operator returns its left-hand operand, the result of the first operator becomes the left-hand operand of the second. As a result, we can chain together output requests. Thus, our statement

  std::cout<<"Enter two numbers:"<<std::endl;

  is equivalent to (std::cout << "Enter two numbers:") << std::endl;
- Each operator in the chain has the same object as its left-hand operand, in this case std::cout. Alternatively, we can generate the same output using two statements:

  ```
  std::cout << "Enter two numbers:";
  std::cout << std::endl;
  ```
- cin is an object of istream class. It stands as an alias for the console input device i.e., the keyboard.
- Input operator (>>) originally right shift operator, has had its definition extended in C++. In the given context, it operates as the extraction operator also called get from operator.
- It is a binary operator. The operand on its left must be some object of istream class. The operand on its right must be a variable of some fundamental data type.
- The value for the variable on RHS of the extraction operator is extracted from the stream originating from the device associated with the object on its left.
- Like the output operator, the input operator returns its left-hand operand as its result. Hence, the statement std::cin >> v1 >> v2;

  is equivalent to(std::cin >> v1) >> v2;
- Because the operator returns its left-hand operand, we can combine a sequence of input requests into a single statement.
- Our input operation reads two values from std::cin, storing the first in v1 and the second in v2. In other words, our input operation executes as

  ```
  std::cin >> v1;
  std::cin >> v2;
  ```

```
//add.cpp

#include<iostream>

int main()

{

        std::cout<<"Enter two numbers:"<<std::endl;

        int num1,num2;

        std::cin>>num1>>num2;

        std::cout<<"The sum of "<<num1<<" and "<<num2<<" is 
"<<num1+num2<<std::endl;

        return 0;

}
```

Output:

```
Enter two numbers:
2 3
The sum of 2 and 3 is 5
```

#include<iostream> tells the compiler that we want to use the iostream library. The name inside angular brackets is a header.

The first statement in the body of main executes an expression. In C++ an expression yields a result and is composed of one or more operands and usually an operator. The expression in this statement uses the output operator (the « operator, also called, insertion operator, put to operator) to print a message on the standard output:

```
std::cout << "Enter two numbers:" << std::endl;
```

endl is a special value called a manipulator. Writing endl has the effect of ending the current line and flushing the buffer associated with that device. Flushing the buffer ensures that all the output the program has generated so far is actually written to the output stream, rather than sitting in memory waiting to be written.

Warning: Programmers often add print statements during debugging. Such statements should always flush the stream. Otherwise, if the program crashes, output may be left in the buffer, leading to incorrect inferences about where the program crashed.

Introduction to Namespaces - Avoiding Pollution of Global Namespace

Consider a situation, when we have two persons with the same name, XYZ, in the same class. Whenever we need to differentiate them definitely we would have to use some additional information along with their name, like either the area, if they live in different area or their mother's or father's name, etc.

Same situation can arise in your C++ applications. For example, you might be writing some code that has a function called xyz() and there is another library available which is also having same function xyz(). Now the compiler has no way of knowing which version of xyz() function you are referring to within your code.

A namespace is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries. Using namespace, you can define the context in which names are defined. In essence, a namespace defines a scope. Namespaces help avoid inadvertent name clashes. The names defined by the C++ library are in the namespace std.

- A namespace definition begins with the keyword namespace followed by the namespace name as follows −

```
namespace namespace_name {
            // code declarations
}
```

- To call the namespace-enabled version of either function or variable, prepend (::) the namespace name as follows –

```
name::code            // code could be variable or function
```

The using directive

You can also avoid prepending of namespaces with the using namespace directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace.
Subsequent code can refer to cout without prepending the namespace, but other items in the std namespace will still need to be explicit as follows −

```
#include <iostream>
using std::cout;
int main () {
        cout << "Hello World " << std::endl;
        return 0;
}
```

The following usage of “using” lets you to refer all items in std namespace without prepending namespace std.

//helloworld.cpp

```
        #include <iostream>
        using namespace std;
        int main () {
                cout << " Hello World " << endl;
                return 0;
        }
```

Output: HelloWorld

//namespace1.cpp

```
#include<iostream>
using namespace std;
//declare and define your own namespace
namespace A
{
int a = 23;
}
int main()
{
 int a = 45;
 //use :: (scope) operator to access the variable defined in namespace A
 cout<<A::a<<endl<<a<<endl;
 return 0;
}
```

Output:

```
23
45
```

//namespace2.cpp

```
#include <iostream>
using namespace std;

// first name space
namespace first_space {
  void func() {
    cout << "Inside first_space" << endl;
  }
}
// second name space
namespace second_space {
  void func() {
    cout << "Inside second_space" << endl;
  }
}

int main () {
  // Calls function from first name space
  first_space::func();

  // Calls function from second name space
  second_space::func();

  return 0;
}
```

Output:

```
Inside first_space
Inside second_space
```

//namespace3.cpp

```
#include <iostream>
using namespace std;

// first name space
namespace first_space {
  void func() {
    cout << "Inside first_space" << endl;
  }
}
// second name space
namespace second_space {
  void func() {
    cout << "Inside second_space" << endl;
  }
}

using namespace first_space;
int main () {
  // This calls function from first name space
  func();
  return 0;
}
```

Output:

```
Inside first_space
```

Constants and Variables

Variables: (Rules same as in C language)

Variables can be declared anywhere inside a function and not necessarily at its very beginning.

```
//varScope.cpp
//Program for illustration purposes only
//It is a bad style for a function to use a global variable and then
//a local variable with the same name
#include<iostream>
using namespace std;

int a=5;

int main(void)
{
        cout<<"a="<<a<<endl;
        int a=10;
        cout<<"Now a="<<a<<endl;        //local scope has higher precedence over global
scope
}
```

Output:

```
a=5
Now a=10
```

Constants: defined using keyword **const**.

```
//const.cpp
#include<iostream>
using namespace std;

int main()
{
	const int a=5;
	cout<<a<<endl;

	//a=5;				//Error, because a is defined to be a constant
	cout<<a<<endl;
}
```

Output:

```
5
5
```

```
//const1.cpp
//Constants have to be initialized when defined
#include<iostream>
using namespace std;
int main()
{
	int const a=10;
	const int b=20;

//Naturally, a const variable must be initialized when defined
//	const int c;		//Error: uninitialized const
//	int const d;		//Error: uninitialized const

	cout<<"const a="<<a<<endl<<"const b="<<b<<endl;

//	The value of a const variable cannot be changed after definition
//	Error: assignment of read-only variable   a=50
//	a=50;


//	Error: invalid conversion from const int* to int*
//	int *ptr2;
//	ptr2=&a;
}
```

```
Output:
const a=10
const b=20
```

//const2.cpp

```
//Pointer to constant data (pointee (pointed data) cannot be changed)
#include<iostream>
using namespace std;

int main()
{
	const int a=10;			//a is constant
	int b=20;
	const int* ptr1=&b;		//ptr1 is not constant, it points to constant data

//	a is constant and cannot be changed
//	a=100;
	cout<<"a="<<a<<endl;


//	ptr1 points to constant data(b) and cannot be changed
//	*ptr1=200;
	cout<<"b="<<b<<" "<<"ptr1 points to b: "<<*ptr1<<endl;
//	b is not constant, therefore can be changed
	b=200;
	cout<<"b="<<b<<" "<<"ptr1 points to b: "<<*ptr1<<endl;


//	pointer may point to some other variable, because ptr1 is not constant
	int c=30;
	ptr1=&c;
	cout<<"c="<<c<<" "<<"ptr1 points to c: "<<*ptr1<<endl;

	c=300;		//Okay, because c is not constant
	cout<<"c="<<c<<" "<<"ptr1 points to c: "<<*ptr1<<endl;

//Error: ptr1 points to constant data(c), which cannot be changed
//	*ptr1=300;
}
```

```
Output:
a=10
b=20 ptr1 points to b:20
b=200 ptr1 points to b:200
c=30 ptr1 points to c:30
```

```
c=300 ptr1 points to c:300
```

```
//const3.cpp
//Constant pointer
#include<iostream>
using namespace std;

int main()
{
        int m = 4, n = 5;
//      Error
//      int * const p;
        int * const p = &n;             //p is a constant pointer pointing to n

        cout<<"n="<<n<<" p points to n: "<<*p<<endl;
//      Error: p cannot refer to any other variable
//      p=&m;

        n=50;
        cout<<"n="<<n<<" p points to n: "<<*p<<endl;

        *p=500;
        cout<<"n="<<n<<" p points to n: "<<*p<<endl;
}
```

Output:

```
n=5 p points to n: 5
n=50 p points to n: 50
n=500 p points to n: 500
```

User Defined Function: Function Call Mechanism, Function Overloading – Static Polymorphism, Function Call Resolution

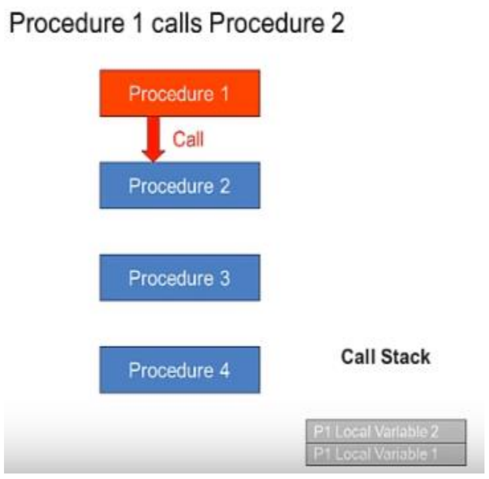
Function Call Mechanism:

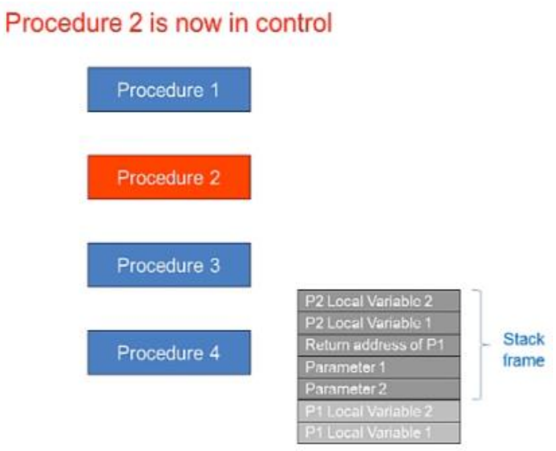
Call Stack:

Consider 4 functions Procedure1, Procedure2, Procedure3, and Procedure4. Let each of them have 2 Parameters.

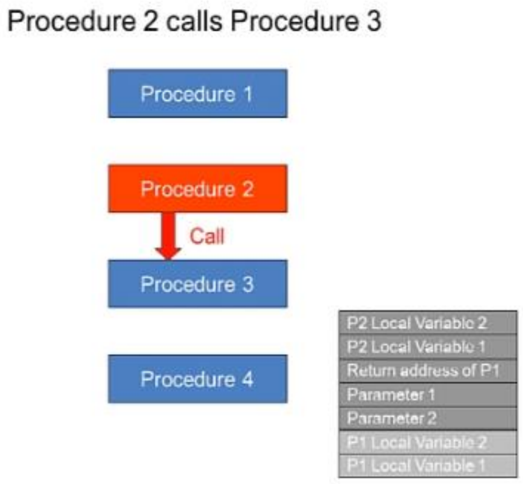
Procedure 1 gives call to procedure 2, which in turn calls procedure 3, which in turn calls procedure 4.

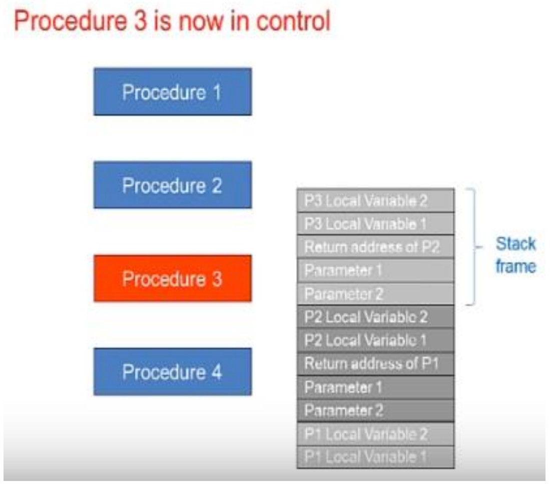
When procedure 1 calls procedure 2, the parameters which procedure 2 is expecting is pushed on to the stack in reverse order (i.e., parameter2 then parameter 1). Next Return address of procedure 2 is pushed on to the stack. This comes from the CPU's Program Counter. Finally any local variables are also pushed on to the stack. So procedure 2 is now in control and its stack frame is in place. So it can get a hold of any data it needs.





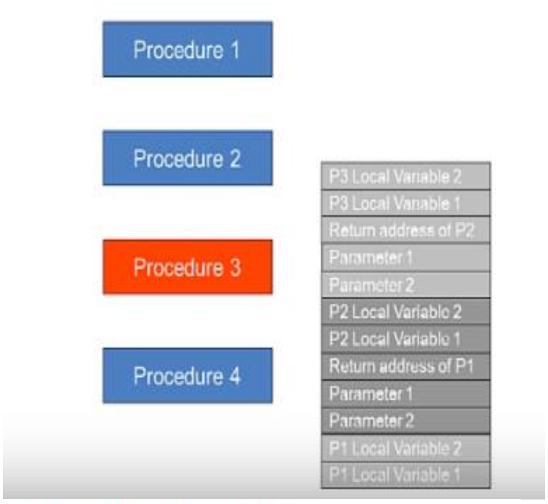
Next, when procedure 2 calls procedure 3. The same steps happen. Procedure 3 is now in control.



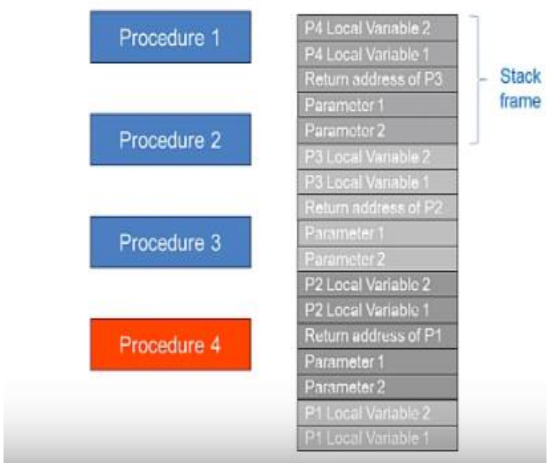


Next, when procedure 3 calls procedure 3. The same steps happen. Procedure 4 is now in control.

Procedure 3 calls Procedure 4

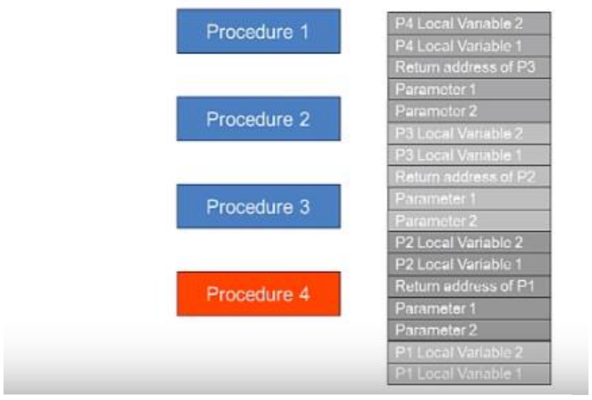


Procedure 4 is now in control

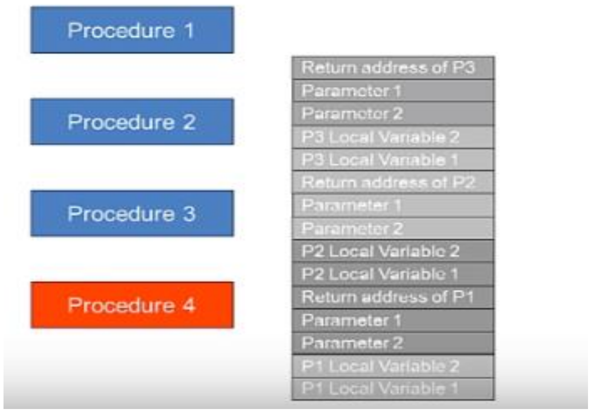


Now, when procedure 4 is finished, the stack will be torn down. Procedure 4 now returns control to procedure 3. Any local variables of procedure 4 are popped out of the stack.

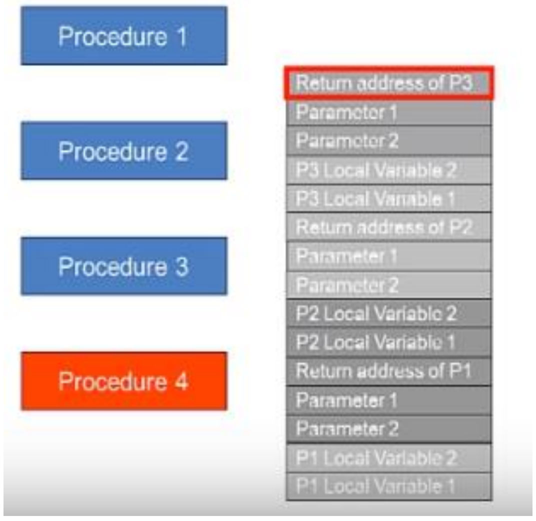
Procedure 4 returns control to Procedure 3

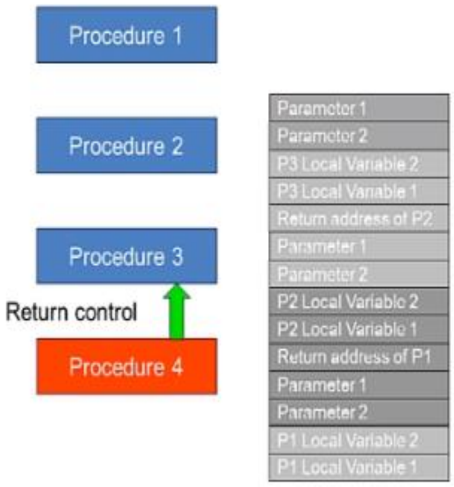


Pops items to obtain return address

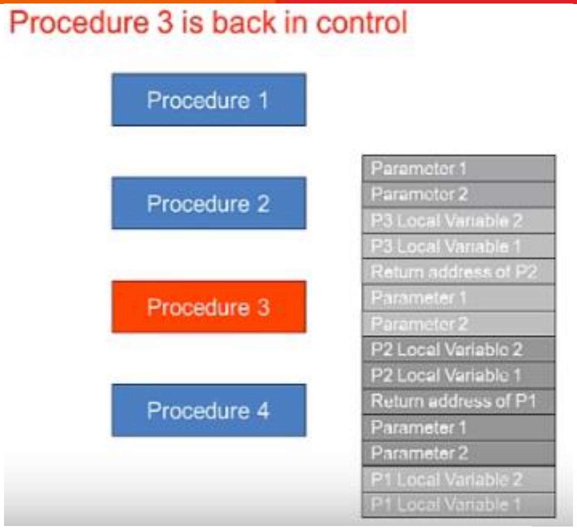


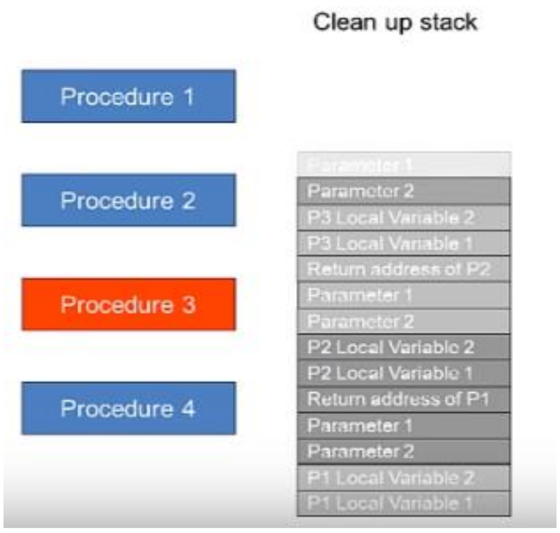
Procedure 4 can then pop off the return address of procedure 3 to repopulate the program counter and return control to procedure 3.

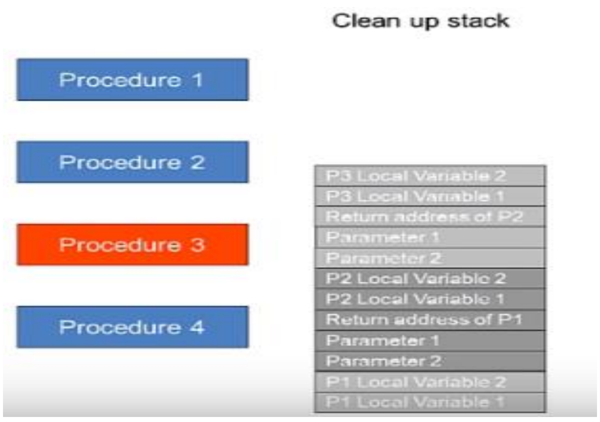


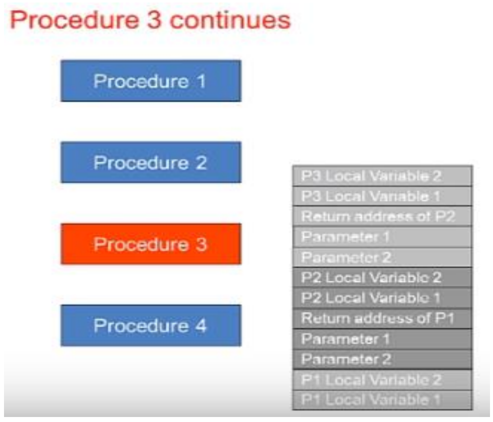


In this scheme, procedure 3 is responsible for cleaning up the rest of procedure 4 stack frame. So procedure 3 is now back in control; it cleans up the stack and can pick from where it left off.









The same process continues for the other procedures.

Function Overloading – Static Polymorphism

- Functions that have the same name but different parameter lists and that appear in the same scope are overloaded. In other words, define multiple functions having the same name but different signature (Function name, type of parameter, sequence of parameters).

Note: Two functions having the same signature but different return types will result in a compilation error due to an attempt to re-declare.

- Overloading allows Static Polymorphism (Binding happens at compile time) i.e., Function selection (or Function matching or Overload Resolution) is performed by the compiler based on the number and the types of the actual parameters at the places of invocation during compile time.
- Function overloading eliminates the need to invent—and remember—names that exist only to help the compiler figure out which function to call.

Note: The main function may not be overloaded.

Consider a database application with several functions to find a record based on name, phone number, account number, and so on. Function overloading lets us define a collection of functions, each named lookup, that differ in terms of how they do the search. We can call lookup passing a value of any of several types:

```
Record lookup(const Account&);  // find by Account
Record lookup(const Phone&);    // find by Phone
Record lookup(const Name&);     // find by Name
```

```
Account acct; Phone phone;
Record r1 = lookup(acct);          // call version that takes an Account
Record r2 = lookup(phone);         // call version that takes a Phone
```

Here, all three functions share the same name, yet they are three distinct functions. The compiler uses the argument type(s) to figure out which function to call.

- Same no. of parameters but different data types

//addOverload.cpp

```
#include<iostream>
using namespace std;

int add(int,int);
double add(double,double);
```

```
int main()
{
	cout<<add(2,2)<<endl;
	cout<<add(2.5,2.6)<<endl;
}
int add(int a,int b)
{
	cout<<"2 int functions"<<endl;
	return a+b;
}

double add(double a,double b)
{
	cout<<"2 doublefunctions"<<endl;
	return a+b;
}
```

Output:

```
2 int functions
4
2 double functions
5.1
```

- Different no. of parameters

```
//areaOverload.cpp
#include<iostream>
using namespace std;

double area(double);
double area(double,double);

int main()
{
    double side=2;
    cout<<"Area of square "<<area(side)<<endl;          //calls area(double)

    double length=2,breadth=3;
    cout<<"Area of rectangle "<<area(length,breadth)<<endl; //calls area(double,double)
}
double area(double a)
{
        return a*a;
}
double area(double l,double b)
{
        return l*b;
}
```

Output:

```
Area of square 4
Area of rectangle 6
```

3 steps in Overload Resolution

1) Finding candidate functions - Same name as that of the call and for which declaration is visible at the point of call

Note: If there are no candidate functions then the call is an error.

2) Finding viable functions from candidate functions -

   a) The function must have the same number of parameters as there are arguments in the call.
   b) The type of each argument must match - or be convertible to - the type of its corresponding parameter.

Note: If there are no viable functions then the call is an error.

3) Finding the best match, if any

   a) An exact match
      - Argument and parameter types are the same
      - Argument is converted from an array or function type to corresponding pointer type
   b) Match through a const conversion
   c) Match through a promotion
   d) Match through an arithmetic or pointer conversion
   e) Match through a class type conversion (discussed later)

Note: If there is no best match then the call is an error

Array to pointer conversion

```
  int ar[10];
```

```
void f(int *a);
Call: f(ar);
```

Conversion to const

Non const objects can be converted to const

```
int a=2;     void f(const int* ptr);
Call: f(&a);
```

Examples of promotion

- char to int
- float to double
- enum to int/short/unsigned int/…
- bool to int

Arithmetic Conversion

- Rules define a hierarchy of type conversions in which operands to an operator are converted to the widest type

Pointer Conversion

- A constant integral value of 0 and the literal nullptr can be converted to any pointer type
- A pointer to any nonconst type can be converted to void*
- A pointer to any type can be converted to a const void*

Question: Apply Overload Resolution, to the following functions:

In the context of a list of function prototypes:

```
int g(double);                   // F1
void f();                        // F2
void f(int);                     // F3
double h(void);                  // F4
int g(char, int);                // F5
void f(double, double = 3.4);    // F6
void h(int, double);        // F7
void f(char, char *);       // F8
```

• The call site to resolve is:

```
f(5.6);
```

Solution:

- Resolution:

Candidate functions (by name): F2, F3, F6, F8
Viable functions (by # of parameters): F3, F6
Best viable function (by type double { Exact Match}): F6

```
//overloadAmbiguity1.cpp
//This program doesn’t compile
#include<iostream>
using namespace std;

void f(int,int);                        //F1
void f(double,double);                  //F2

int main()
{
        f(2,3.5);  //matches both F1 and F2, therefore ambiguous call, compile error
                        //Both F1 and F2 require 2 conversions
        //f(2.3,(double)5);   //Ambiguity may be resolved by explicit typecast during
fn call
}

void f(int a,int b)
{
        cout<<"2 int fn"<<endl;
}
void f(double a,double b)
{
        cout<<"2 double fn"<<endl;
}

//add1Overload.cpp
//matches the function with least no. of conversions
/#include<iostream>
using namespace std;
int add(int,int);
//double add(double,double);
int add(int,double);
int main()
{
	cout<<add(2,2)<<endl;
	cout<<add(2.5,2.5)<<endl;
	cout<<add(2.5,2)<<endl;
}
int add(int a,int b)
{
	cout<<"2 int function"<<endl;
	return a+b;
}
/*
double add(double a,double b)
{
	cout<<"2 double function"<<endl;
	return a+b;
}
*/
int add(int a,double b)
{
	cout<<" int,double function"<<endl;
	return a+b;
}
```

Output:

```
2 int function
4
int,double function
4
2 int function
4
```

//Overload Ambiguity: This program doesn't compile
// overloadAmbiguity2.cpp

```
#include<iostream>
using namespace std;
int add(int a) {return a;}                              //F1
int add(int a,int b=0) {return a+b;}                    //F2
int add(int a,int b,int c=0) {return a+b+c;}            //F3

int main()
{
        int x=10,y=20,z=30;
        add(10);                            //ambiguity: matches F1, F2
        add(10,20);                         //ambiguity: matches F2, F3
        add(10,20,30);                      //OK
}
```

Default Parameters (or Default Arguments)

Some functions have parameters that are given a particular value in most, but not all, calls. In such cases, we can declare that common value as a default argument for the function. Functions with default arguments can be called with or without that argument.

- Default values are specified while prototyping the function
- Default parameters are required while calling functions with fewer arguments or without any argument
- Better to use default value for less used parameters

For example, we might use a string to represent the contents of a window. By default, we might want the window to have a particular height, width, and background character. However, we might also want to allow users to pass values other than the defaults. To accommodate both default and specified values we would declare our function to define the window as follows:

```
typedef string::size_type sz;        // typedef
string screen(sz ht = 24, sz wd = 80, char backgrnd = ' ');
```

Here we've provided a default for each parameter. A default argument is specified as an initializer for a parameter in the parameter list. We may define defaults for one or more parameters. However, if a parameter has a default argument, all the parameters that follow it must also have default arguments.

Calling Functions with Default Arguments

If we want to use the default argument, we omit that argument when we call the function. Because screen provides defaults for all of its parameters, we can call screen with zero, one, two, or three arguments:

```
string window;
window = screen();              // equivalent to screen(24,80,' ')
window = screen(66);            // equivalent to screen(66,80,' ')
window = screen(66, 256);       // screen(66,256,' ')
window = screen(66, 256, '#');  // screen(66,256,'#')
```

Arguments in the call are resolved by position. The default arguments are used for the trailing (right-most) arguments of a call. For example, to override the default for background, we must also supply arguments for height and width:

```
window = screen(, , '?'); // error: can omit only trailing arguments
window = screen('?');     // calls screen('?',80,' ')
```

Note that the second call, which passes a single character value, is legal. Although legal, it is unlikely to be what was intended. The call is legal because '?' is a char, and a char can be converted to the type of the left-most parameter. That parameter is string::size_type, which is an unsigned integral type. In this call, the char argument is implicitly converted to string::size_type, and is passed as the argument to height. On our machine, '?' has the hexadecimal value 0x3F, which is decimal 63. Thus, this call passes 63 to the height parameter.

Note: Part of the work of designing a function with default arguments is ordering the parameters so that those least likely to use a default value appear first and those most likely to use a default appear last.

Default Argument Declarations

Although it is normal practice to declare a function once inside a header, it is legal to redeclare a function multiple times. However, each parameter can have its default specified only once in a given scope. Thus, any subsequent declaration can add a default only for a parameter that has not previously had a default specified. As usual, defaults can be specified only if all parameters to the right already have defaults. For example, given

```
// no default for the height or width parameters
string screen(sz, sz, char = ' ');

//we cannot change an already declared default value:
string screen(sz, sz, char = '*');          // error: redeclaration

//but we can add a default argument as follows:
string screen(sz = 24, sz = 80, char);   // ok: adds default arguments to ht and wd
```

Best Practices

Default arguments ordinarily should be specified with the function declaration in an appropriate header.

Default Argument Initializers

Local variables may not be used as a default argument. Excepting that restriction, a default argument can be any expression that has a type that is convertible to the type of the parameter:

```
// the declarations of wd, def, and ht must appear outside a function
sz wd = 80;
char def = ' ';
sz ht( );
string screen(sz = ht( ), sz = wd, char = def);                 //Function Prototype

string window = screen();                              //Function call, calls screen(ht( ), 80,
' ')
```

Names used as default arguments are resolved in the scope of the function declaration. The value that those names represent is evaluated at the time of the call:

```
void f2( )
{
   def =   '*';             // changes the value of a default argument
   sz wd = 100;                  // hides the outer definition of wd but does not change
the default
   window = screen( );        // calls screen(ht( ), 80, '*')
}
```

Inside f2, we changed the value of def. The call to screen passes this updated value. Our function also declared a local variable that hides the outer wd. However, the local named wd is unrelated to the default argument passed to screen.

Example Program:

```
//defaultParamLocalCheck.cpp
#include <iostream>
using namespace std;

int wd=30;     int ht =20;      char ch='/';
void f2();
void defaultCheck(int=wd,int=ht,char=ch);

int main()
{
   defaultCheck();
   f2();
   return 0;
}
void f2()
{
   int wd=35;
   ht=25;
   defaultCheck();
}

void defaultCheck(int wd,int ht,char ch)
{
   cout<<"wd= "<<wd<<" ht= "<<ht<<" ch= "<<ch<<endl;
}
```

Output:

```
wd= 30 ht= 20 ch= /
wd= 30 ht= 25 ch= /
```

Exercise:

1) Which, if either, of the following declarations are errors? Why?
 (a) int ff(int a, int b = 0, int c = 0);
 (b) char *init(int ht = 24, int wd, char bckgrnd);

2) Which, if any, of the following calls are illegal? Why? Which, if any, are legal but unlikely to match the programmer's intent? Why?
 char *init(int ht, int wd = 80, char bckgrnd = ' ');
 (a) init( );
 (b) init(24,10);
 (c) init(14, '*');

```
//default1.cpp
#include<iostream>
using namespace std;
int sum(int,int y=2);
int main()
{
	cout<<sum(2)<<endl;		//Calls sum(2,2) with default value 2 for 2nd
parameter
	cout<<sum(2,3)<<endl;		//Calls sum(2,3)
}
int sum(int x,int y)
{
	return x+y;
}
```

Output:

```
4
5
```

Restriction on default parameters

- All parameters to the right of a parameter with default argument must have default arguments (function f)
- Default arguments cannot be re-defined (function g)
- All non-defaulted parameters needed in a call (call g())

```
//default2.cpp
void f(int,double=0.0,char);				// Error: ??

int main()
{
int i=5;
double d=1.2;
char ch='a';
f(i,ch);
return 0;
}
```

default parameters: a special case of function overload

```
//default3_1.cpp
#include<iostream>
using namespace std;

void g(int,double,char='a');
void g(int,double=0.0,char);			//OK, a new overload
void g(int=0,double,char);		//OK, a new overload
int main()
{
	int i=5; double d=1.2;char c='b';
	g();				//Prints: 0 0 a
```

```
        g(i);                           //Prints: 5 0 a
        g(i,d);                         //Prints: 5 1.2 a
        g(i,d,c);                       //Prints: 5 1.2 b
        return 0;
}
void g(int i,double d,char c)
{
        cout<<i<<" "<<d<<" "<<c<<endl;
}
//default3_2.cpp
#include<iostream>
using namespace std;

void f(int a,int b=2);
//void f(int a,int b=3);        //Error, redeclaration, because b is already given default value
void f(int a=5,int b);          //Not a redeclaration, because a is given default value this time
                                //Also, b is already given a default value in previous declaration
int main()
{
        f( );                   //calls f(5,2)
        f(50);                  //calls f(50,2)
        f(50,20);               //calls f(50,20)
        return 0;
}

void f(int a,int b)
{
        cout<<a<<" "<<b<<endl;
}
```

Output:

```
5 2
50 2
50 20
```

Default Parameter & Function Overload

• Function overloading can use default parameter

• However, with default parameters, the overloaded functions should still be resolvable.

//defaultAndOverload.cpp

```
#include<iostream>
using namespace std;

int Area(int a, int b = 10) { return (a * b); }
double Area(double c, double d) { return (c * d); }
int main() {
int x = 10, y = 12, t;
double z = 20.5, u = 5.0, f;
t = Area(x);                              // Binds int Area(int, int = 10)
cout << "Area = " << t << endl;           // t = 100
f = Area(z, u);                           // Binds double Area(double, double)
cout << "Area = " << f << endl;           // f = 102.5
return 0;
}
```

Output:

```
Area = 100
Area = 102.5
```

Function overloading with default parameters may fail

//overloadAmbiguity3.cpp

```
//Program doesn't compile
#include <iostream>
using namespace std;

int f();              //F1
int f(int = 0);       //F2
int f(int, int);      //F3

int main()
{
	int x = 5, y = 6;
	f();              //Error: Ambiguity between F1 and F2
	f(x);             // int f(int);
	f(x, y);          // int f(int, int);

	return 0;
}
```

Overload Resolution Fails

• Consider the overloaded function signatures:

//overloadAmbiguity4.cpp

```
int fun(float a) {...}                // Function 1
int fun(float a, int b) {...}                   // Function 2
int fun(float x, int y = 5) {...}               // Function 3
int main() {
        float p = 4.5, t = 10.5;
        int s = 30;
        fun(p, s);                // CALL - 1
        fun(t);                   // CALL – 2

        return 0;
}
```

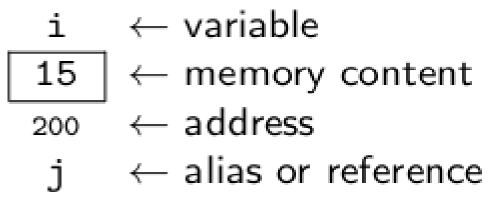
Resolution:
- CALL - 1: Matches Function 2 & Function 3
- CALL - 2: Matches Function 1 & Function 3
- Results in ambiguity

Reference Variable (Alias), Reference Parameters, Garbage and Dangling Reference, Pointers and Reference

A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.

Syntax: `<datatype> &<ref_var_name> = <existing_var_name>;`

```
int i = 15;             // i is a variable
int &j = i;             // j is a reference to i
```



Behavior of Reference

```
//ref1.cpp
#include <iostream>
using namespace std;
int main()
{
        int a = 10, &b = a;             // b is reference of a

// a and b have the same memory
```

```
        cout << "a = " << a << ", b = " << b << endl;
        cout << "&a = " << &a << ", &b = " << &b << endl;

// Changing a appears as change in b
        ++a;
        cout << "a = " << a << ", b = " << b << endl;

// Changing b also changes a
        ++b;
        cout << "a = " << a << ", b = " << b << endl;
}
```

Output:

```
a = 10, b = 10
&a = 0x7ffd027b344c, &b = 0x7ffd027b344c
a = 11, b = 11
a = 12, b = 12
```

Note: a and b have the same memory location and hence the same value. Changing one changes the other and vice-versa.

Pitfalls in Reference

Wrong declaration	Reason	Correct declaration
`int& i;`	no variable to refer to – must be initialized	`int& i = j;`
`int& j = 5;`	no address to refer to as 5 is a constant	`const int& j = 5;`
`int& i = j + k;`	only temporary address (result of `j + k`) to refer to	`const int& i = j + k;`

```
//ref2.cpp
#include<iostream>
using namespace std;

void fnParamTest(int,int &);

int main()
{
        int a=2;

        cout << "a = " << a << ", &a = " << &a << endl;
        fnParamTest(a,a);
}

void fnParamTest(int b,int &c)
```

```
{
	//b: value parameter, c: reference parameter
	cout << "b = " << b << ", &b = " << &b << endl;
	cout << "c = " << c << ", &c = " << &c << endl;
}
```

Output:
```
a = 2, &a = 0x7ffe0d07ec64
b = 2, &b = 0x7ffe0d07ec3c
c = 2, &c = 0x7ffe0d07ec64
```

Param b is call-by- value while param c is call-by- reference
Actual param a and formal param b get the same value in called function
Actual param a and formal param c get the same value in called function
Actual param a and formal param c get the same address in called function
However, actual param a and formal param b have different addresses in called function

```
//ref3.cpp
/* if a reference is passed and the ref var changes its value in called fn, its affect is seen in existing variable in calling function also */
#include<iostream>
using namespace std;

void fn(int &);

int main()
{
	int a=2;

	cout<<"a= "<<a<<endl;
	fn(a);
	cout<<"a= "<<a<<endl;
}
void fn(int &iRef)
{
	iRef++;
}
```
Output:
```
a=2
a=3
```

//ref4.cpp

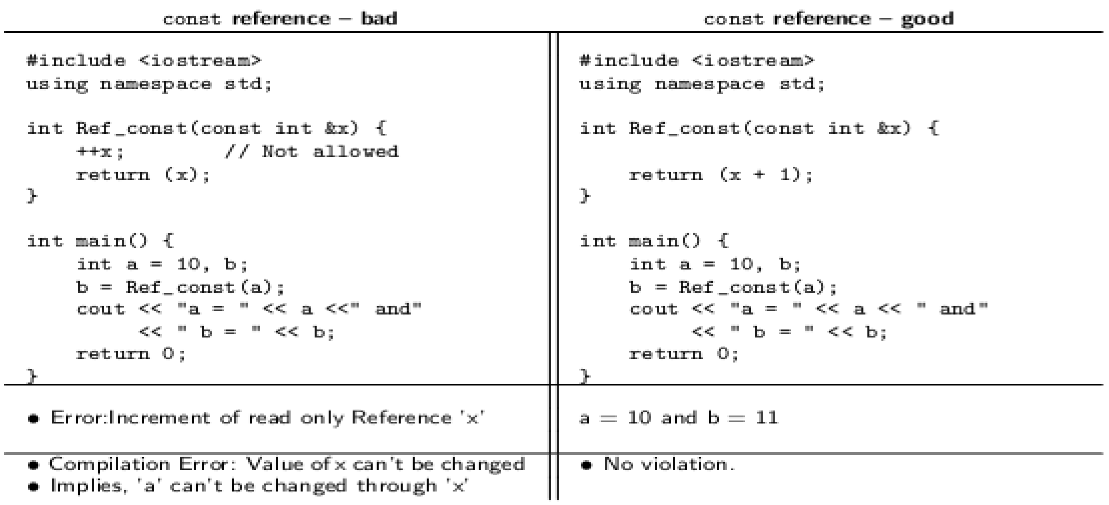
/* Avoid returning the reference of a local variable. After the function returns, the local variable goes out of scope. The reference here becomes a **dangling reference**. Therefore accessing unallocated block of memory may lead to run time errors. Garbage is an object that cannot be reached through a reference.*/

```
#include<iostream>
using namespace std;

int& abc()
{
        int x=10;
        cout<<"x="<<x<<" &x="<<&x<<endl;
        return x;
}
int main()
{
        int &res=abc();
        cout<<"res="<<res<<" &res="<<&res<<endl;
        return 0;
}
/*Run time error:
x=10 &x=0x7ffdb0a46c54
Segmentation fault (core dumped)*/
```

Reference Parameter as const

- A reference parameter may get changed in the called function
- Use const to stop reference parameter being changed



const reference – bad	const reference – good

`#include <iostream>`
`using namespace std;`
 	
`int Ref_const(const int &x) {`
`    ++x;          // Not allowed`
`    return (x);`
`}`
 	
`int main() {`
`    int a = 10, b;`
`    b = Ref_const(a);`
`    cout << "a = " << a <<" and"`
`         << " b = " << b;`
`    return 0;`
`}`
	`#include <iostream>`
	`using namespace std;`
	
	`int Ref_const(const int &x) {`
	
	`    return (x + 1);`
	`}`
	
	`int main() {`
	`    int a = 10, b;`
	`    b = Ref_const(a);`
	`    cout << "a = " << a << " and"`
	`         << " b = " << b;`
	`    return 0;`
	`}`

• Error:Increment of read only Reference 'x'	a = 10 and b = 11
• Compilation Error: Value of x can't be changed • Implies, 'a' can't be changed through 'x'	• No violation.

//Return by reference

```
#include <iostream>
```

```
using namespace std;
int& Return_ref(int &x)
{
        return (x);
}
int main( )
{
        int a = 10, b;
        b = Return_ref(a);
        cout << "a = " << a << " and b = "<< b << endl;
        Return_ref(a) = 3;
        cout << "a = " << a<<endl;
        return 0;
}
```

Output:

```
a = 10 and b = 10
a=3
```

Note how a value is assigned to a function call.
This can change a local variable

Note: When returning a reference, be careful that the object being referred to does not go out of scope. So it is not legal to return a reference to local var. But you can always return a reference on a static variable.

Pointer and Reference

Pointer	**Reference**
Refers to an address	Refers to an address
Pointers can point to NULL. int *p = NULL; // p is not pointing	References cannot be NULL int &j ; //wrong

Pointers can point to different variables at different times
int a, b, *p;
p = &a; // p points to a
...
p = &b // p points to b
	For a reference, its referent is fixed
	int a, c, &b = a; // Okay
	....
	&b = c // Error

Allows users to operate on the address diff pointers, increment, etc	Does not allow users to operate on the address. All operations are interpreted for the referent
Array of pointers can be defined	Array of references not allowed

/*Note: There is a template function by name swap in the standard library
To avoid calling of the generic swap function present in standard library, here in all swap programs, I have written the **using** statements separately for cout and endl, instead of writing **using namespace std */**

//Call By Value
//swapCallbyValue.cpp

```
#include<iostream>
using std::cout;
using std::endl;

void swap(int,int);

int main()
{
	int a=2,b=3;
	cout<<"Before swap: "<<"a="<<a<<"  b="<<b<<endl;
	swap(a,b);
	cout<<"After swap: "<<"a="<<a<<"  b="<<b<<endl;
}

void swap(int m,int n)
{
	int temp;
	temp=m;
	m=n;
	n=temp;
}
```

Output:

```
Before swap: a=2  b=3
After swap: a=2  b=3
```

//Call By pointer/address
//swapCallbyPtr.cpp

```
#include<iostream>
using std::cout;
using std::endl;

void swap(int *,int *);

int main()
{
	int a=2,b=3;
	cout<<"Before swap: "<<"a="<<a<<"  b="<<b<<endl;
	swap(&a,&b);
	cout<<"After swap: "<<"a="<<a<<"  b="<<b<<endl;
}

void swap(int *m,int *n)
{
	int temp;
	temp=*m;
	*m=*n;
	*n=temp;
}
```

Output:

```
Before swap: a=2  b=3
After swap: a=3  b=2
```

//Call By Reference

```
#include<iostream>
using std::cout;
using std::endl;

void swap(int &,int &);
int main()
{
	int a=2,b=3;
	//char a='c',b='d';
	//double a=2.5,b=3.5;
	cout<<"Before swap: "<<"a="<<a<<"  b="<<b<<endl;
	swap(a,b);
	cout<<"After swap: "<<"a="<<a<<"  b="<<b<<endl;
}

void swap(int &m,int &n)
{
	int temp;
	temp=m;
	m=n;
	n=temp;
}
```

Output:

```
Before swap: a=2  b=3
After swap: a=3  b=2
```

//pointer-pointee computation (without const)

//ptrPointee.cpp

```
#include<iostream>
using namespace std;

int main()
{
        int a=10;
        int *ptr1=&a;

        cout<<"a="<<a<<" "<<"*ptr1="<<*ptr1<<endl;

        a=20;
        cout<<"a="<<a<<" "<<"*ptr1="<<*ptr1<<endl;

        *ptr1=30;
        cout<<"a="<<a<<" "<<"*ptr1="<<*ptr1<<endl;

        int c=40;
        ptr1=&c;
        cout<<"a="<<a<<"  "<<"*ptr1 now points to c="<<*ptr1<<"     c="<<c<<endl;
}
```

Output:

```
a=10 *ptr1=10
a=20 *ptr1=20
a=30 *ptr1=30
a=30  *ptr1 now points to c=40     c=40
```

/*value of const var cannot be chngd after defn. so const data cannot be initialized to a pointer pointing to non const data*/

//constVarToNonConstPtr.cpp

```
#include<iostream>
using namespace std;

int main()
{
        const int a=10;
        //Error
        int *ptr=&a;
}
```

Pointers and Dynamic Allocation

In C++, dynamic memory is managed through a pair of operators: **new**, which allocates, and optionally initializes, an object in dynamic memory(from free store) and returns a pointer to that object; and **delete**, which takes a pointer to a dynamic object, destroys that object, and frees the associated memory.

Dynamic memory is problematic because it is surprisingly hard to ensure that we free memory at the right time. Either we forget to free the memory—in which case we have a memory leak—or we free the memory when there are still pointers referring to that memory—in which case we have a pointer that refers to memory that is no longer valid.

- Operator new for allocation on free store
- No size specification needed, type suffices
- Allocated memory returned as int *
- No casting needed
- Can be initialized
- Operator delete for de-allocation from free store
- Core language feature { no header needed}

```
//dynamicMemory.cpp
#include<iostream>
using namespace std;
int main()
{
//      int *p=new int;
//      *p=5;
//The above 2 statements is equivalent to below statement i.e., initialization is also possible
        int *p=new int(5);
        cout<<*p<<endl;
        delete p;
        return 0;
}
```

Output:

```
5
```

```
//dynamicMemoryArray.cpp
/*Allocation by operator new[] (different from operator new) on free store
```

```
# of elements explicitly passed to operator new[]
Release by operator delete[] (different from operator delete) from free store*/
#include<iostream>
using namespace std;

int main()
{
        int *a=new int[5];

        for(int i=0;i<5;i++)
        {
                //cin>>a[i];                    //May read input from user
                a[i]=i;
        }
        cout<<"The numbers are:"<<endl;
        for(int i=0;i<5;i++)
                cout<<a[i]<<" ";
        cout<<endl;

        delete[] a;
        return 0;
}
```

Output:

```
The numbers are:
0 1 2 3 4
```

//Usage of pointer notation

```
#include<iostream>
using namespace std;
int main()
{
        int *a=new int[5];

        for(int i=0;i<5;i++)
        {
                //cin>>*(a+i);          //may read input from user
                *(a+i)=i;
        }
        cout<<"The numbers are:"<<endl;
        for(int i=0;i<5;i++)
                cout<<*(a+i)<<" ";
        cout<<endl;
        delete[] a;
        return 0;
}
```

```
//To showcase initialization of array allocated dynamically
#include<iostream>
using namespace std;

int main()
{
        int *a=new int[5]{1,2,3,4,5};        //Initialization of dynamically allocated memory

        cout<<"The numbers are:"<<endl;
        for(int i=0;i<5;i++)
                cout<<a[i]<<" ";
        cout<<endl;

        delete[] a;
        return 0;
}
```

Note: Allocation and De-Allocation must correctly match. Do not free the space created by new using free(). And do not use delete if memory is allocated through malloc(). These may results in memory corruption.

Allocator	De-allocator
malloc	free
operator new	operator delete
operator new [ ]	operator delete [ ]

Efficiency and Flexibility: Inline Function (Efficiency), Template Function (Flexibility)

Inline Function

Consider the following function:

```
const string &shorterString(const string &s1, const string &s2)
{
        return s1.size() <= s2.size() ? s1 : s2;
}
```

shorterString is a small function that returns a reference to the shorter of its two string parameters. The benefits of defining a function for such a small operation include the following:

- It is easier to read and understand a call to shorterString than it would be to read and understand the equivalent conditional expression.
- Using a function ensures uniform behavior. Each test is guaranteed to be done the same way.
- If we need to change the computation, it is easier to change the function than to find and change every occurrence of the equivalent expression.
- The function can be reused rather than rewritten for other applications.

There is, however, one potential drawback to making shorterString a function: Calling a function is apt to be slower than evaluating the equivalent expression. On most machines, a function call does a lot of work: Registers are saved before the call and restored after the return; arguments may be copied; and the program branches to a new location.

inline Functions Avoid Function Call Overhead

A function specified as inline (usually) is expanded "in line" at each call. If shorterString were defined as inline, then this call

```
cout << shorterString(s1, s2) << endl;
```

(probably) would be expanded during compilation into something like

```
cout << (s1.size() < s2.size() ? s1 : s2) << endl;
```

The run-time overhead of making shorterString a function is thus removed. We can define shorterString as an inline function by putting the keyword **inline** before the function's return type:

```
// inline version: find the shorter of two strings
inline const string &shorterString(const string &s1, const string &s2)
{
        return s1.size( ) <= s2.size( ) ? s1 : s2;
}
```

Note: The inline specification is only a request to the compiler. The compiler may choose to ignore this request.

In general, the inline mechanism is meant to optimize small, straight-line functions that are called frequently. Many compilers will not inline a recursive function. A 75-line function will almost surely not be expanded inline.

//inlineFn.cpp

```
#include<iostream>
using namespace std;

inline int sqr(int x){return x*x;}
```

```
int main()
{
	cout<<"sqr(2)= "<<sqr(2)<<endl;
}
```

Output:

```
sqr(2)= 4
```

Note: preprocessor macro: Preprocessor facility that behaves like an inline function.

Template Function

Templates are the foundation for generic programming. Template is a blueprint or formula for creating classes or functions. When we use a generic type, such as vector, or a generic function, such as find, we supply the information needed to transform that blueprint into a specific class or function. That transformation happens during compilation.

C++ adds two new keywords to support templates: ‘template’ and ‘typename’. The second keyword can always be replaced by keyword ‘class’. A function template is a formula from which we can generate type-specific versions of that function. Instead of writing swap function separately for int, double, char, we may write one function and pass datatype as parameter.

- A template definition starts with the keyword template followed by a template parameter list, which is a comma-separated list of one or more template parameters bracketed by the less-than (<) and greater-than (>) tokens.
- The template parameter list acts much like a function parameter list. A function parameter list defines local variable(s) of a specified type but does not say how to initialize them. At run time, arguments are supplied that initialize the parameters. Analogously, template parameters represent types or values used in the definition of a class or function. When we use a template, we specify—either implicitly or explicitly —template argument(s) to bind to the template parameter(s).
- Our swap function declares one type parameter named T. Inside swap, we use the name T to refer to a type. Which actual type T represents is determined at compile time based on how swap is used.
- For eg: swap(2,3) the arguments have type int. The compiler will deduce int as the template argument and will bind that argument to the template parameter T.

//templateFnSwap.cpp

```
#include<iostream>
using namespace std;

template<typename T>                    //or template<class T>
void swapG(T &a,T &b)
{
        T temp;
        temp=a;
        a=b;
        b=temp;
}
int main()
{
        int int1=2,int2=3;
        cout<<"Integer Swap"<<endl;
        cout<<"Before Swap: "<<int1<<" "<<int2<<endl;
        swapG<int>(int1,int2);
        cout<<"After Swap: "<<int1<<" "<<int2<<endl;

        double double1=2.3,double2=3.2;
        cout<<endl<<"Double Swap"<<endl;
        cout<<"Before Swap: "<<double1<<" "<<double2<<endl;
        swapG<double>(double1,double2);
        cout<<"After Swap: "<<double1<<" "<<double2<<endl;

        char ch1='a',ch2='b';
        cout<<endl<<"Character Swap"<<endl;
        cout<<"Before Swap: "<<ch1<<" "<<ch2<<endl;
        swapG<char>(ch1,ch2);
        cout<<"After Swap: "<<ch1<<" "<<ch2<<endl;

        return 0;
}
```

Output:

```
Integer Swap
Before Swap: 2 3
After Swap: 3 2
```

```
Double Swap
Before Swap: 2.3 3.2
After Swap: 3.2 2.3
Character Swap
Before Swap: a b
After Swap: b a
```

Type Inference

- Type Inference refers to automatic deduction of the data type of an expression in a programming language.
- **auto** keyword specifies that the type of the variable that is being declared will be automatically deduced from its initializer.
- In case of functions, if their return type is auto then that will be evaluated by return type expression at runtime.
- **typeid** operator is used where dynamic type of an object needs to be known.
- **typeid(x).name( )** returns shorthand name of the data type of x. The actual returned name is compiler dependent.
- on g++, it returns i for int, d for double, f for float, c for char, Pi for pointer to int, Pd for pointer to double, etc..

//cPP11auto.cpp

```
#include<iostream>
#include<typeinfo>
using namespace std;
int main()
{
        auto a=1;
        auto b=1.5;
        auto c=1.5f;
        auto d='x';
        auto e="s";
        auto f=&a;
        auto g=&b;
        cout<<"a="<<a<<"  type="<<typeid(a).name()<<endl;
        cout<<"b="<<b<<"  type="<<typeid(b).name()<<endl;
        cout<<"c="<<c<<"  type="<<typeid(c).name()<<endl;
        cout<<"d="<<d<<"  type="<<typeid(d).name()<<endl;
```

```
	cout<<"e="<<e<<"  type="<<typeid(e).name()<<endl;
	cout<<"f="<<f<<"  type="<<typeid(f).name()<<endl;
	cout<<"g="<<g<<"  type="<<typeid(g).name()<<endl;
	//Below line is an error, because witout initialization the data type cannot be deduced
	//auto h;
	return 0;
}
/*Compilation: g++ -std=c++11 pgm1_cPP11auto.cpp
Output: ./a.out
a=1  type=i
b=1.5  type=d
c=1.5  type=f
d=x  type=c
e=s  type=PKc
f=0x7ffe48f15e80  type=Pi
g=0x7ffe48f15e88  type=Pd
*/
```

Lambda Function

A lambda expression represents a callable unit of code. It can be thought of as an unnamed, inline function. Like any function, a lambda has a return type, a parameter list, and a function body. Unlike a function, lambdas may be defined inside a function.

A lamba expression has the form:

[capture list] (parameter list) -> return type **{ function body }**

Where capture list is an (often empty) list of local variables defined in the enclosing function; return type, parameter list, and function body are the same as in any ordinary function. However, unlike ordinary functions, a lambda must use a **trailing return** to specify its return type.

We can omit either or both of the parameter list and return type but must always include the capture list and the function body.

Note: Lambdas with function bodies that contain anything other than a single return statement that does not specify a return type returns void.

Lambda expressions allow you to write anonymous functions inline, removing the need to write a separate function or a function object. Lambda expressions can make code easier to read and understand.

Syntax

Let’s start with a very simple lambda expression. The following example defines a lambda expression that just writes a string to the console. A lambda expression starts with square brackets [], followed by curly braces, {}, which contain the body of the lambda expression.

The lambda expression is assigned to the **basicLambda** auto-typed variable. The second line executes the lambda expression using normal function call syntax.

```
//Lambda is a C++ 11 feature, on g++ compile using g++ -std=c++11 lambda1.cpp
//lambda1.cpp
#include<iostream>
using namespace std;
int main()
{
        auto basicLambda=[]{cout<<"Hello From Lambda\n"; return 0;};
        basicLambda();
        return 0;
}
```

Output: Hello From Lambda

A lambda expression can accept a parameter. Parameters are specified between parentheses and separated by commas, just as with normal functions.

```
//lambda2.cpp
#include<iostream>
using namespace std;

int main()
{
        auto parametersLambda = [](int value){ cout << "The value is " << value << 
endl; };
        parametersLambda(42);
        return 0;
}
```

Output:

```
The value is 42
```

A lambda expression can return a value. The return type is specified following an arrow, called a trailing return type. The following example defines a lambda expression accepting two parameters and returning the sum:

//lambda3a.cpp

```
#include<iostream>
using namespace std;
int main()
{
        auto returningLambda = [](int a, int b) -> int { return a + b; };
        int sum = returningLambda(11, 22);
        cout<<"sum="<<sum<<endl;
        return 0;
}
```

Output:

```
sum=33
```

The return type can be omitted even if the lambda expression does return something. If the return type is omitted, the compiler deduces the return type of the lambda expression according to the same rules as for function return type deduction. In the previous example, the return type can be omitted as follows:

//lambda3b.cpp

```
#include<iostream>

using namespace std;

int main()

{

        auto returningLambda = [](int a, int b){ return a + b; };

        int sum = returningLambda(11, 22);

        cout<<"sum="<<sum<<endl;

        return 0;

}
```

Output:

```
sum=33
```

A lambda expression can capture variables from its enclosing scope. For example, the following lambda expression captures the variable data so that it can be used in its body.

```
double data = 1.23;

auto capturingLambda = [data]{ cout << "Data = " << data << endl; };
```

The square brackets part is called the lambda capture block. It allows you to specify how you want to capture variables from the enclosing scope. Capturing a variable means that the variable becomes available inside the body of the lambda expression. Specifying an empty capture block, [], means that no variables from the enclosing scope are captured. When you just write the name of a variable in the capture block as in the preceding example, then you are capturing that variable by value.

```
//lambda4.cpp
#include<iostream>
using namespace std;
int main()
{
	double data = 1.23;
	auto capturingLambda = [data]
	{
	//Below line is an error: increment of read-only variable 'data'
	//	data++;
		cout << "Data = " << data << endl;
	};
	capturingLambda();
	return 0;
}
```

Output:

```
Data=1.23
```

The compiler transforms a lambda expression into some kind of functor. The captured variables become data members of this functor. Variables captured by **value** are copied into data members of the functor. These data members have the same constness as the captured variables. In the preceding capturingLambda example, the functor gets a non-const data member called data, because the captured variable, data, is non-const. However, in the following example, the functor gets a const data member called data, because the captured variable is const.

```
const double data = 1.23;

auto capturingLambda = [data]{ cout << "Data = " << data << endl; };
```

A functor always has an implementation of the function call operator. **For a lambda expression, this function call operator is marked as const by default**. That means that **even if you capture a non-const variable by value in a lambda expression, the lambda expression will not be able to modify the copy.** You can mark the function call operator as non-const by specifying the lambda expression as **mutable** as follows:

```
double data = 1.23;

auto capturingLambda =  [data] () mutable { data *= 2; cout << "Data = " << data << endl;
};
```

In this example, the non-const variable data is captured by value, thus the functor gets a non-const data member that is a copy of data. Because of the mutable keyword, the function call operator is marked as non-const, thus the body of the lambda expression can modify its copy of data. **Note that if you specify mutable, then you have to specify the parentheses for the parameters even if they are empty.**

```
//lambda5.cpp

#include<iostream>

using namespace std;

int main()

{

        double data = 1.23;

        auto capturingLambda=[data]() mutable{data=data*2; cout<< "Data = " << data << endl; };

        capturingLambda();

        cout<<"Data outside lambda = "<<data<<endl;

        return 0;

}
```

Output:

```
Data = 2.46

Data outside lambda = 1.23
```

You can prefix the name of a variable with **&** to capture it by **reference**. The following example captures the variable 'data' by reference so that the lambda expression can directly change data in the enclosing scope:

```
double data = 1.23;

auto capturingLambda=[&data]()mutable{data=data*2; cout << "Data = " << data << endl; };
```

Note: When you capture a variable by reference, you have to make sure that the reference is still valid at the time the lambda expression is executed.

//lambda6.cpp

```
#include<iostream>
using namespace std;
int main()
{
        double data = 1.23;
        auto capturingLambda=[&data]()mutable{data=data*2; cout<<"Data = "<< data<<
endl; };
        capturingLambda();
        cout<<"Data outside lambda = "<<data<<endl;
        return 0;
}
```

Output:

```
Data = 2.46
Data outside lambda = 2.46
```

There are two ways to capture all variables from the enclosing scope:

[=] captures all variables by value

[&] captures all variables by reference

It is also possible to selectively decide which variables to capture and how, by specifying a capture list with an optional capture default. Variables prefixed with & are captured by reference. Variables without a prefix are captured by value. The capture default should be the first element in the capture list and be either & or =. Here are some capture block examples:

[&x] captures only x by reference and nothing else.

[x] captures only x by value and nothing else.

[=, &x, &y] captures by value by default, except variables x and y, which are captured by reference.

[&, x] captures by reference by default, except variable x, which is captured by value.

[&x, &x] is illegal because identifiers cannot be repeated.

[this] captures the surrounding object. In the body of the lambda expression you can access this object, even without using this->.

WARNING: It is not recommended to capture all variables from the enclosing scope with [=], [&], or with a capture default, even though it is possible. Instead, you should selectively capture only what's needed.

Full Syntax

The complete syntax of a lambda expression is as follows:

[capture_block](parameters) mutable exception_specification attribute_specifier

-> return_type {body}

A lambda expression contains the following parts:

Capture block: specifies how variables from the enclosing scope are captured and made available in the body of the lambda.

Parameters: (optional) a list of parameters for the lambda expression. You can omit this list only if you do not need any parameters and you do not specify mutable, an exception specification, attribute specifier, or a return type. The parameter list is similar to the parameter list for normal functions.

Mutable: (optional) marks the lambda expression as mutable;

exception_specification: (optional) can be used to specify which exceptions the body of the lambda expression can throw.

attribute_specifier: (optional) can be used to specify attributes for the lambda expression.

return_type: (optional) the type of the returned value. If this is omitted, the compiler deduces the return type according to the same rules as for function return type deduction.

Note: The capture list is used for local nonstatic variables only; lambdas can use local statics and variables declared outside the function directly.

Attributes are a mechanism to add optional and/or vendor-specific information into source code. Before C++11, the vendor decided how to specify that information. Examples are __attribute__ , __declspec , and so on. Since C++11, there is support for attributes by using the double square brackets syntax [[attribute]] .

The C++11 standard defines only two standard attributes: [[noreturn]] and [[carries_dependency]] . C++14 adds the [[deprecated]] attribute.

****

https://stackoverflow.com/questions/7627098/what-is-a-lambda-expression-in-c11

What is a lambda expression?

A lambda expression, sometimes also referred to as a lambda function or (strictly speaking incorrectly, but colloquially) as a lambda, is a simplified notation for defining and using an anonymous function object. Instead of defining a named class with an operator(), later making an object of that class, and finally invoking it, we can use a shorthand.

When would I use one?

This is particularly useful when we want to **pass an operation as an argument to an algorithm**. In the context of graphical user interfaces (and elsewhere), such operations are often referred to as callbacks.

What class of problem do they solve that wasn't possible prior to their introduction?

Here i guess every action done with lambda expression can be solved without them, but with much more code and much bigger complexity. Lambda expression is the way of optimization for your code and a way of making it more attractive. As said by Stroustup : effective way of optimizing.

Examples with STL Algorithms

Note: Students may refer the document on vector, given at last to know about basic on vectors.

The below examples demonstrates usage of lambdas with the STL algorithm: count_if()

```
//lambdaSTL1a.cpp

//count_if() returns the number of elements in a range that satisfy the condition.

//In this pgm count_if() declared in algorithm takes a lambda function as 3rd argument

//to determine the count of even numbers in the array 'arr'

#include<iostream>

#include<algorithm>

using namespace std;

int main()

{

	int arr[]={10,1,3,6};

	int n=sizeof(arr)/sizeof(arr[0]);

	int cnt = count_if(arr, arr+n, [](int i){ return (i % 2)==0; });

	cout << "Found " << cnt << " even numbers " << endl;

	return 0;

}
```

Output:

```
Found 2 even numbers
```

The following example uses the count_if() algorithm to count the number of elements in the given vector that satisfy a certain condition. The condition is given in the form of a lambda expression, which captures the 'value' variable from its enclosing scope by value.

//lambdaSTL1b.cpp

```cpp
//count_if() returns the number of elements in a range that satisfy the condition.
//In this pgm count_if() declared in algorithm takes a lambda function as 3rd argument
//to determine the count of values in the vector 'vec' which are greater than 3
#include<iostream>
#include<algorithm>
using namespace std;
int main()
{
        vector<int> vec{ 10, 2, 30, 44, 1, 6, 70, 8, 3 };
        int value = 3;
        int cnt = count_if(vec.begin(), vec.end(),  [value](int i){ return i > value; });
        cout << "Found " << cnt << " values > " << value << endl;
        return 0;
}
```

Output:

```
Found 6 values > 3
```

The example can be extended to demonstrate capturing variables by reference. The following lambda expression counts the number of times it is called by incrementing a variable in the enclosing scope that is captured by reference:

//lambdaSTL1c.cpp

```cpp
//count_if() returns the number of elements in a range that satisfy the condition.
//In this pgm count_if() declared in algorithm takes a lambda function as 3rd argument
//to determine the count of values in the vector 'vec' which are greater than 3.
```

//Also it captures cntLambdaCalled by reference to determine the no. of times lambda is called

```
#include<iostream>

#include<algorithm>

using namespace std;

int main()

{

        vector<int> vec{ 10, 2, 30, 44, 1, 6, 70, 8, 3 };

        int value = 3;

        int cntLambdaCalled = 0;

        int cnt = count_if(vec.begin(), vec.end(),

        [value,&cntLambdaCalled](int i){ ++cntLambdaCalled; return i > value; });

        cout << "The lambda expression was called " << cntLambdaCalled<< " times." << endl;

        cout << "Found " << cnt << " values > " << value << endl;

        return 0;

}
```

Output:

```
The lambda expression was called 9 times

Found 6 values > 3
```

transform()

The transform function takes three iterators and a callable. The first two iterators denote an input sequence and the third is a destination. The algorithm calls the given callable on each element in the input sequence and writes the result to the destination. As in this call, the destination iterator can be the same as the iterator denoting the start of the input. When the input iterator and the destination iterator are the same, transform replaces each element in the input range with the result of calling the callable on that element.

In this call, we passed a lambda that returns the absolute value of its parameter. The lambda body is a single return statement that returns the result of a conditional expression. We need not specify the return type, because that type can be inferred from the type of the conditional operator.

```
//lambdaSTL2.cpp

#include<iostream>

#include<algorithm>

using namespace std;

int main()

{

        vector<int> vi={-1,-2,-5,8};

        transform(vi.begin(), vi.end(), vi.begin(), [](int i) { return i < 0 ? -i : i; });

        for (auto i : vi) {

                cout << i << " ";

        }

        cout<<endl;

        return 0;

}
```

Output: 1 2 5 8

Note: Lambda expressions are most useful for simple operations that we do not need to use in more than one or two places. If we need to do the same operation in many places, we should usually define a function rather than writing the same lambda expression multiple times. Similarly, if an operation requires many statements, it is ordinarily better to use a function.

**************************UNIT 1 Complete**************************

Appendix:

Vector in C++ STL

Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators. In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes there may be a need of extending the array. Removing the last element takes only constant time because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time.

Certain functions associated with the vector are:

Iterators

begin() – Returns an iterator pointing to the first element in the vector

end() – Returns an iterator pointing to the theoretical element that follows the last element in the vector

rbegin() – Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element

rend() – Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

cbegin() – Returns a constant iterator pointing to the first element in the vector.

cend() – Returns a constant iterator pointing to the theoretical element that follows the last element in the vector.

crbegin() – Returns a constant reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element

crend() – Returns a constant reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

Capacity

size() – Returns the number of elements in the vector.

max_size() – Returns the maximum number of elements that the vector can hold.

capacity() – Returns the size of the storage space currently allocated to the vector expressed as number of elements.

resize(n) – Resizes the container so that it contains ‘n’ elements.

empty() – Returns whether the container is empty.

shrink_to_fit() – Reduces the capacity of the container to fit its size and destroys all elements beyond the capacity.

reserve() – Requests that the vector capacity be at least enough to contain n elements.

Element access

reference operator [g] – Returns a reference to the element at position 'g' in the vector

at(g) – Returns a reference to the element at position 'g' in the vector

front() – Returns a reference to the first element in the vector

back() – Returns a reference to the last element in the vector

data() – Returns a direct pointer to the memory array used internally by the vector to store its owned elements.

Modifiers

assign() – It assigns new value to the vector elements by replacing old ones

push_back() – It push the elements into a vector from the back

pop_back() – It is used to pop or remove elements from a vector from the back.

insert() – It inserts new elements before the element at the specified position

erase() – It is used to remove elements from a container from the specified position or range.

swap() – It is used to swap the contents of one vector with another vector of same type. Sizes may differ.

clear() – It is used to remove all the elements of the vector container

emplace() – It extends the container by inserting new element at position

emplace_back() – It is used to insert a new element into the vector container, the new element is added to the end of the vector

predicate Function: Function that returns a type that can be converted to bool Often used by the generic algorithms to test elements. Predicates used by the library are either unary (taking one argument) or binary (taking two).

Further Reading(Optional):

https://web.mst.edu/~nmjxv3/articles/lambdas.html

References:

- "C++ Primer", Stanley Lippman, Josee Lajoie, Barbara E Moo, Addison-Wesley Professional, 5th Edition.
- Online resources

Note: In some programs I have commented the lines which may cause error, students are advised to uncomment those lines and check the error message

Any suggestions are welcome.

Acknowledgement:
Dr. Shylaja S S, Professor, CSE, Director – CCBD, PESU
Dr. Partha Pratim Das, IIT Kharagpur, NPTEL Lectures