UNIT IV

TEMPLATES AND EXCEPTION HANDLING

UNIT IV - TEMPLATES AND EXCEPTION HANDLING

- **Topics to be discussed,**
  - Function Template and Class Template
  - Namespaces
  - Casting
  - Exception Handling

UNIT IV - TEMPLATES AND EXCEPTION HANDLING

- **Topics to be discussed,**
  - **Function Template and Class Template**
  - Namespaces
  - Casting
  - Exception Handling

- Need same functionality but it should work for different data types, what concept you use?
  - **Function overloading**

Function Overloading

```
#include<iostream>
using namespace std;
int add(int x,int y)
{
    return (x+y);
}
float add(float x,float y)
{
    return (x+y);
}
int main()
{
    int int_sum;
    float float_sum;
    int_sum=add(12,78);
    cout<<"Integer sum:"<<int_sum<<endl;
    float_sum=add(34.789f,89.456f);
    cout<<"Float sum:"<<float_sum<<endl;

}
```

Output:

```
Integer sum:90
Float sum:124.245
```

Generics in C++

- Generic Programming enables the programmer to write a general algorithm which will work with all data types
- It eliminates the need to create different algorithms for different data types such as integer, string or a character.
- The advantages of Generic Programming are
  - Code Reusability
  - Avoid Function Overloading
  - Once written it can be used for multiple times and cases.
- **Generics can be implemented in C++ using Templates**.

Function Template and Class Template

- **Templates** are the **foundation of generic programming**
- **Templates** are powerful features of C++ which allows us to write **generic** programs.
- It involves writing code in a way that is independent of any particular type.
- A template is a **blueprint** or **formula** for creating a **generic class** or a **function**.
- In simple terms, we can create a single function or a class to work with different data types using templates.
  - For example, if we are writing a function to sort a sequence of data but we don't know what types of data will be passed to that function so we may need to write different functions for different data types, but rather than writing the same for function multiple times for different types we can use the Templates in C++ to make our sort function generic which would sort data of any type.

Function Template and Class Template – Cont'd

- Templates are often used in larger **codebase** for the purpose of code **reusability** and **flexibility** of the programs.
- The concept of templates can be used in two different ways:
  - **Function Templates**
  - **Class Templates**
- For using the templates tool in C++ we must know two keywords: **'template'** and **'typename'**, the typename keyword can be replaced with the keyword **'class'**.

Function Template and Class Template – Cont'd

C++ Function Template

- **We can create a single function to work with different data types by using a template.**
- **Defining a Function Template:**
  - A function template starts with the keyword template followed by template parameter(s) inside <> which is followed by the function definition.

```
template <typename T>
T functionName(T parameter1, T parameter2, ...)
{
   // code
}
```

  - In the above code, **T is a template argument** that accepts different data types (int, float, etc.), and **typename is a keyword**.
  - When an argument of a data type is passed to functionName(), the compiler generates a new version of functionName() for the given data type.

Function Template and Class Template – Cont'd

C++ Function Template

- **Calling a Function Template**
  - Once we've declared and defined a function template, we can call it in other functions or templates (such as the main() function) with the following syntax

```
functionName<dataType>(parameter1, parameter2,...);
```

C++ Function Template – Example 1

```
#include<iostream>
using namespace std;
template<typename T>
T add(T x,T y)
{
    return (x+y);
}
int main()
{
    int int_sum;
    float float_sum;
    int_sum=add<int>(12,78); // calling with int parameters
    cout<<"Integer sum:"<<int_sum<<endl;
    float_sum=add<float>(34.789,89.456);// calling with float parameters
    cout<<"Float sum:"<<float_sum<<endl;

}
```

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

Output:

```
Integer sum:90
Float sum:124.245
```

Function Template and Class Template – Cont'd

- **How Do Templates Work?**
  - Templates are a type of static (compile time) polymorphism
  - They expand at the compilation time.
  - This is similar to macros, the only difference is that the compiler does type-checking before the expansion of the template.
  - In simple words, the source code contains only a function/class but compiled code may contain multiple copies of the same function/class.

```
#include<iostream>

template<typename T>
T add(T num1, T num2) {
  return (num1 + num2);
}

int main() {
  ... ... ...

  result1 = add<int>(12,78);
  ... ... ...

  result2 = add< float > (34.789,89.456);
  ... ... ...

}
```

Compiler internally generates and adds below code

```
int add(int num1, int num2) {
    return (num1 + num2);
}
```

Compiler internally generates and adds below code

```
float add( float num1, float num2) {
  return (num1 + num2);
}
```

- The process of creating functions (with specific types) from function templates (with template types) is called **function template instantiation (or instantiation)**
- When a function is instantiated due to a function call, it's called **implicit instantiation**.
- A function that is instantiated from a template is technically called a **specialization**, but in common language is often called a **function instance**.
- The template from which a specialization is produced is called a **primary template**.

A function call without using angled brackets for same type of data

```
#include<iostream>
using namespace std;
template<typename T>
T add(T x,T y)
{
    return (x+y);
}
int main()
{
    int int_sum;
    float float_sum;
    int_sum=add(12,78); // will instantiate add(int, int)
    cout<<"Integer sum:"<<int_sum<<endl;
    float_sum=add(34.789,89.456);  // will instantiate add(double, double)
    cout<<"Float sum:"<<float_sum<<endl;

}
```

Output:

```
Integer sum:90
Float sum:124.245
```

A function call without using angled brackets for different types of data

```
18  #include<iostream>
19  using namespace std;
20  template<typename T>
21  T add(T x,T y)
22  {
23      return (x+y);
24  }
25  int main()
26  {
27      int int_sum;
28      float float_sum;
29      int_sum=add(12,7.8);
30      cout<<"Integer sum:"<<int_sum<<endl;
31      float_sum=add(34.789,89);
32      cout<<"Float sum:"<<float_sum<<endl;
33
34  }
35
```

ers

::Blocks | Search results | Cccc | Build log | Build messages | CppCheck/Vera++

	Line	Message
		=== Build file: "no target" in "no project" (compiler: unknown) ===
4\CS320...		In function 'int main()':
4\CS320...	29	error: no matching function for call to 'add(int, double)'
4\CS320...	21	note: candidate: 'template<class T> T add(T, T)'
4\CS320...	21	note: template argument deduction/substitution failed:
4\CS320...	29	note: deduced conflicting types for parameter 'T' ('int' and 'double')
4\CS320...	31	error: no matching function for call to 'add(double, int)'
4\CS320...	21	note: candidate: 'template<class T> T add(T, T)'
4\CS320...	21	note: template argument deduction/substitution failed:
4\CS320...	31	note: deduced conflicting types for parameter 'T' ('double' and 'int')
		=== Build failed: 2 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===

A function call without using angled brackets for different types of data

- In our function call add(12, 7.8), we're passing arguments of two different types: one int and one double.
- Because we're making a function call without using angled brackets to specify an actual type, the compiler will first look to see if there is a non-template match for max(int, double).
- It won't find one.
- Next, the compiler will see if it can find a function template match
- However, this will also fail, for a simple reason: T can only represent a single type.
- There is no type for T that would allow the compiler to instantiate function template add<T>(T, T) into a function with two different parameter types.
- Put another way, because both parameters in the function template are of type T, they must resolve to the same actual type.

A function call without using angled brackets for different types of data

```
#include<iostream>
using namespace std;
template<typename T>
T add(T x,T y)
{
    return (x+y);
}
int main()
{
    int int_sum;
    float float_sum;
    int_sum=add(12,static_cast<int>(7.8));
    cout<<"Integer sum:"<<int_sum<<endl;
    float_sum=add(34.789,static_cast<double>(89));
    cout<<"Float sum:"<<float_sum<<endl;

}
```

Output:

```
Integer sum:19
Float sum:123.789
```

A function call without using angled brackets for different types of data

```
#include<iostream>
using namespace std;
template<typename T,typename U>
T add(T x,U y)
{
    return (x+y);
}
int main()
{
    cout<<"sum1:"<<add(12,7.8)<<endl;
    cout<<"sum2:"<<add(7.8,12)<<endl;
    cout<<"sum3:"<<add(12.0,7.8)<<endl;
}
```

Output:

```
sum1:19
sum2:19.8
sum3:19.8
```

A function call without using angled brackets for different types of data

```
#include<iostream>
using namespace std;
template<typename T,typename U>
U add(T x,U y)
{
    return (x+y);
}
int main()
{
    cout<<"sum1:"<<add(12,7.8)<<endl;
    cout<<"sum2:"<<add(7.8,12)<<endl;
    cout<<"sum3:"<<add(12.0,7.8)<<endl;
}
```

Output:

```
sum1:19.8
sum2:19
sum3:19.8
```

A function call without using angled brackets for different types of data

```
#include<iostream>
using namespace std;
template<typename T,typename U>
auto add(T x,U y)
{
    return (x+y);
}
int main()
{
    cout<<"sum1:"<<add(12,7.8)<<endl;
    cout<<"sum2:"<<add(7.8,12)<<endl;
    cout<<"sum3:"<<add(12.0,7.8)<<endl;
}
```

auto return type -- we'll let the compiler deduce what the return type should be from the return statement

Output:

```
sum1:19.8
sum2:19.8
sum3:19.8
```

Abbreviated function templates C++20

- C++20 introduces a new use of the auto keyword: When the auto keyword is used as a parameter type in a normal function, the compiler will automatically convert the function into a function template with each auto parameter becoming an independent template type parameter. This method for creating a function template is called an abbreviated function template.

```
#include<iostream>
using namespace std;
auto add(auto x, auto y)
{
    return (x+y);
}
int main()
{
    cout<<"sum1:"<<add(12,7.8)<<endl;
    cout<<"sum2:"<<add(7.8,12)<<endl;
    cout<<"sum3:"<<add(12.0,7.8)<<endl;
}
```

Output:

```
sum1:19.8
sum2:19.8
sum3:19.8
```

is shorthand in C++20 for the following:

```
template <typename T, typename U>
auto max(T x, U y)
{
    return (x+y)
}
```

which is the same as the add function template we wrote before.

C++ Function Template – Example 2

```
#include<iostream>
using namespace std;
template<typename T>
void sortData(T a[],int n)
{
    int i,j;
    T t;
    for(i=0;i<n-1;i++)
    {
        for(j=i+1;j<n;j++)
        {
            if(a[j]<a[i])
            {
                t=a[i];
                a[i]=a[j];
                a[j]=t;
            }
        }
    }
}
```

```
int main()
{
    int n;
    cout<<"\nEnter n:";
    cin>>n;
    int *a=new int[n];
    cout<<"\nEnter integer elements:";
    for(int i=0;i<n;i++)
        cin>>a[i];
    cout<<"\nGiven data:";
    for(int i=0;i<n;i++)
        cout<<a[i]<<" ";
    sortData<int>(a,n);
    cout<<"\nSorted data:";
    for(int i=0;i<n;i++)
        cout<<a[i]<<" ";
    cout<<"\nEnter n:";
    cin>>n;
    float *b=new float[n];
    cout<<"\nEnter floating point data:";
    for(int i=0;i<n;i++)
        cin>>b[i];
    cout<<"\nGiven data:";
    for(int i=0;i<n;i++)
        cout<<b[i]<<" ";
    sortData<float>(b,n);
    cout<<"\nSorted data:";
    for(int i=0;i<n;i++)
        cout<<b[i]<<" ";
}
```

Output:

```
Enter n:4

Enter integer elements:34
56
12
8

Given data:34 56 12 8
Sorted data:8 12 34 56
Enter n:5

Enter floating point data:12.7
67.45
23.78
11.345
55.89

Given data:12.7 67.45 23.78 11.345 55.89
Sorted data:11.345 12.7 23.78 55.89 67.45
```

Create the C++ Function Template named swapNum so that it has two parameters of the same type. A Template Function created from swapNum will exchange the values of these two parameters. (Test for two different types of data)

```
#include<iostream>
using namespace std;
template<typename T>
void swapNum(T &x,T &y)
{
   T t;
   t=x;
   x=y;
   y=t;
}
int main()
{
   int n1,n2;
   cout<<"\nEnter two integers:";
   cin>>n1>>n2;
   cout<<"\nBefore swap:";
   cout<<"\nn1="<<n1<<"\nn2="<<n2;
   swapNum<int>(n1,n2);
   cout<<"\nAfter swap:";
   cout<<"\nn1="<<n1<<"\nn2="<<n2;
   float f1,f2;
   cout<<"\nEnter float values:";
   cin>>f1>>f2;
   cout<<"\nBefore swap:";
   cout<<"\nf1="<<f1<<"\nf2="<<f2;
   swapNum<float>(f1,f2);
   cout<<"\nAfter swap:";
   cout<<"\nf1="<<f1<<"\nf2="<<f2;
}
```

```
Enter two integers:45 78

Before swap:
n1=45
n2=78
After swap:
n1=78
n2=45
Enter float values:89.98 34.786

Before swap:
f1=89.98
f2=34.786
After swap:
f1=34.786
f2=89.98
```

Function Template and Class Template – Cont'd

- **Class Templates**
  - Similar to function templates, we can use class templates **to create a single class to work with different data types**.
  - Class templates come in handy as they can make our code shorter and more manageable.
- **Class Template Declaration**
  - A class template starts with the keyword template followed by template parameter(s) inside <> which is followed by the class declaration.

```
template <class T>
class className
{
  private:
    T var;
    ... .. ...
  public:
    T functionName(T arg);
    ... .. ...
};
```

Function Template and Class Template – Cont'd

- **Creating a Class Template Object**
  - Once we've declared and defined a class template, we can create its objects in other classes or functions (such as the main() function) with the following syntax:

    className<dataType> classObject;

- For example,

  className<int> classObject;

  className<float> classObject;

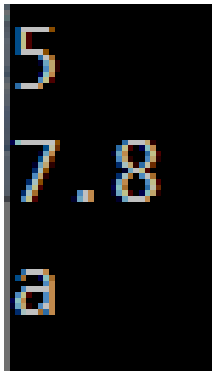
  className<string> classObject;

Class Template - Example

```
#include<iostream>
using namespace std;
template <class T>
class Number
{
   private:
   T n;
   public:
   Number(T x)
   {
      n=x;
   }
   T getVar()
   {
      return n;
   }
};
```

```
int main()
{
   Number <int> n1(5);
   cout<<n1.getVar()<<endl;
   Number <float> n2(7.8);
   cout<<n2.getVar()<<endl;
   Number <char> n3('a');
   cout<<n3.getVar()<<endl;
}
```

Output:



Function Template and Class Template – Cont'd

- **Defining a Class Member Outside the Class Template**
  - Suppose we need to define a function outside of the class template. We can do this with the following code:

```
template <class T>
class ClassName
{
   ... .. ...
   // Function prototype
   returnType functionName();
};
// Function definition
template <class T>
returnType ClassName<T>::functionName()
{
   // code
}
```

Notice that the code template <class T> is repeated while defining the function outside of the class. This is necessary and is part of the syntax.

```
#include <iostream>
using namespace std;
template <class T>
class Calculator
{
  private:
   T num1, num2;
   public:
   Calculator(T n1, T n2)
   {
      num1 = n1;
      num2 = n2;
   }
   void displayResult();
   T add() { return num1 + num2; }
   T subtract() { return num1 - num2; }
   T multiply() { return num1 * num2; }
   T divide() { return num1 / num2; }
};
```

```
template <class T>
 void Calculator<T>:: displayResult()
 {
      cout << "Numbers: " << num1 << " and " << num2 << "." << endl;
      cout << num1 << " + " << num2 << " = " << add() << endl;
      cout << num1 << " - " << num2 << " = " << subtract() << endl;
      cout << num1 << " * " << num2 << " = " << multiply() << endl;
      cout << num1 << " / " << num2 << " = " << divide() << endl;
   }

int main()
{
   Calculator<int> intCalc(6,7);
   Calculator<float> floatCalc(5.6,7.8);
   cout << "Int results:" << endl;
   intCalc.displayResult();
   cout << endl;
   cout<< "Float results:" << endl;
   floatCalc.displayResult();
   return 0;
}
```

Output:

```
Int results:
Numbers: 6 and 7.
6 + 7 = 13
6 - 7 = -1
6 * 7 = 42
6 / 7 = 0

Float results:
Numbers: 5.6 and 7.8.
5.6 + 7.8 = 13.4
5.6 - 7.8 = -2.2
5.6 * 7.8 = 43.68
5.6 / 7.8 = 0.717949
```

Function Template and Class Template – Cont'd

- **C++ Class Templates With Multiple Parameters**
  - In C++, we can use multiple template parameters and even use default arguments for those parameters.
  - For example,

```
template <class T, class U, class V = int>
class ClassName
{
 private:
   T member1;
   U member2;
   V member3;
   ... .. ...
 public:
   ... .. ...
};
```

Class Templates With Multiple Parameters - Example

```
#include <iostream>
using namespace std;
// Class template with multiple and default parameters
template <class T, class U, class V = char>
class ClassTemplate
{
  private:
   T var1;
   U var2;
   V var3;
  public:
   ClassTemplate(T v1, U v2, V v3)
   {
      var1=v1;
      var2=v2;
      var3=v3;
   }
   void printVar()
   {
      cout << "var1 = " << var1 << endl;
      cout << "var2 = " << var2 << endl;
      cout << "var3 = " << var3 << endl;
   }
};
```

```
int main()
{
   // create object with int, double and char types
   ClassTemplate<int, double> obj1(7, 7.7, 'c');
   cout << "obj1 values: " << endl;
   obj1.printVar();

   // create object with int, double and bool types
   ClassTemplate<double, char, bool> obj2(8.8, 'a', false);
   cout << "\nobj2 values: " << endl;
   obj2.printVar();
   return 0;
}
```

Output:

```
obj1 values:
var1 = 7
var2 = 7.7
var3 = c

obj2 values:
var1 = 8.8
var2 = a
var3 = 0
```

UNIT IV - TEMPLATES AND EXCEPTION HANDLING

- **Topics to be discussed,**
  - Function Template and Class Template
  - **Namespaces**
  - Casting
  - Exception Handling

Namespaces in C++

- Consider a situation, when we have two persons with the same name, Vijay, 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.

Namespaces in C++ - Cont'd

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

Namespaces in C++ - Cont'd

- 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, we can define the context in which names are defined.
- In essence, a namespace defines a scope.

Namespaces in C++ - Cont'd

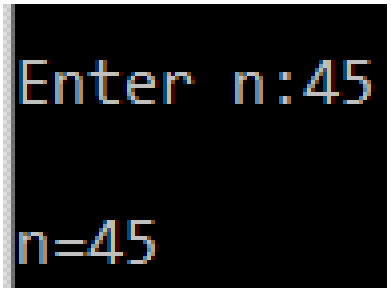
- A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it.
- Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when our code base includes multiple libraries.
- All identifiers at namespace scope are visible to one another without qualification.
- Identifiers outside the namespace can access the members by using the fully qualified name for each identifier

Namespaces in C++ - Cont'd

Example:

```
#include<iostream>
int main()
{
   int n;
   std::cout<<"\nEnter n:";
   std::cin>>n;
   std::cout<<"\nn="<<n;
}
```

Output:



- Notice that we used, std::cin and std::cout instead of cin and cout
- The prefix std:: indicates that the names cin and cout are defined inside the namespace std.

Namespaces in C++ - Cont'd

- Defining a Namespace
  - A namespace definition begins with the keyword **namespace** followed by the namespace name as follows,

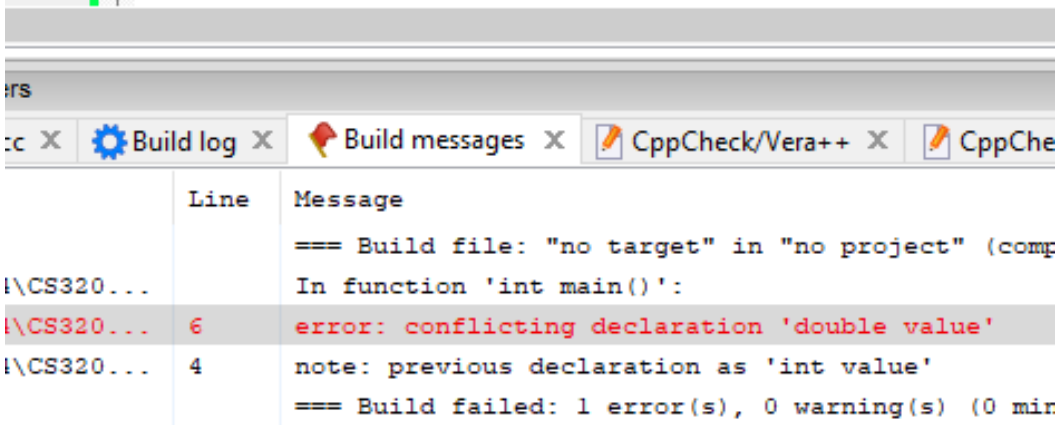
```
namespace namespace_name
 {
// code declarations i.e. variable (int a;)
method (void add();)
classes ( class student{};)
}
```

- *It is to be noted that, there is no semicolon (;) after the closing brace*.
- To call the namespace-enabled version of either function or variable, prepend the namespace name as follows:
  - namespace_name: :code; // code could be variable , function or class.

Namespaces in C++ - Cont'd

- Consider the following C++ program:

```
1  #include<iostream>
2  int main()
3  {
4      int value;
5      value = 0;
6      double value;
7      value = 0.0;
8  }
```



- In each scope, a name can only represent one entity.
- So, there cannot be two variables with the same name in the same scope.
- Using namespaces, we can create two variables or member functions having the same name.

variables with the same name in the different scope - Example

```
#include <iostream>
using namespace std;
// Variable created inside namespace
namespace first
{
      int val = 500;
}
 // Global variable
int val = 100;
int main()
{
   // Local variable
   int val = 200;
   cout<<"val in main="<<val<<endl;
   cout<<"val in Global scope="<<::val<<endl;
   cout << "val in first namespace="<<first::val << endl;
   return 0;
}
```

Output:

```
val in main=200
val in Global scope=100
val in first namespace=500
```

functions with the same name in the different scope - Example

```
#include <iostream>
using namespace std;
namespace first // first name space
{
  void print() {
    cout << "print Inside first namespace" << endl;
  }
}
namespace second // second name space
{
  void print() {
    cout << "print Inside second namespace" << endl;
  }
}
int main ()
 {
  // Calls function from first name space.
  first::print();
  // Calls function from second name space.
  second::print();
  return 0;
}
```

Output:

```
print Inside first namespace
print Inside second namespace
```

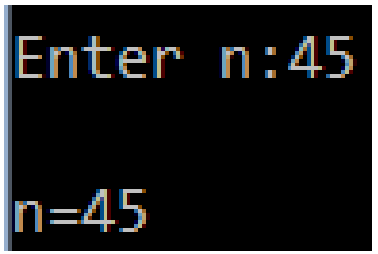
Namespaces in C++ - Cont'd

- **The using directive:**
  - We can 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.

```
#include<iostream>
using std::cout;
int main()
{
    int n;
    cout<<"\nEnter n:";
    std::cin>>n;
    cout<<"\nn="<<n;
}
```

```
#include<iostream>
using namespace std;
int main()
{
    int n;
    cout<<"\nEnter n:";
    cin>>n;
    cout<<"\nn="<<n;
}
```

Output:



```
#include <iostream>
using namespace std;
namespace first // first name space
{
  void print() {
    cout << "print Inside first namespace" << endl;
  }
}
namespace second // second name space
{
  void print() {
    cout << "print Inside second namespace" << endl;
  }
}
using namespace first;
int main ()
 {
  // Calls function from first name space.
  print();
  // Calls function from second name space.
  second::print();
  return 0;
}
```

Output:

```
print Inside first namespace
print Inside second namespace
```

Namespaces in C++ - Cont'd

- **Classes and Namespace**
- As like variables and functions, class can also be defined inside a namespace

Classes in a Namespace - Example

```
#include<iostream>
using namespace std;
 namespace nspc
{
   class Tech // A Class in a namespace
   {
   public:
      void show()
      {
         cout<<"nspc::Tech::show()"<<endl;
      }
   };
}
int main()
{
   nspc::Tech obj;
   obj.show();
   return 0;
}
```

Output:

```
nspc::Tech::show()
```

A class can also be declared inside namespace and defined outside namespace

```
#include<iostream>
using namespace std;
 namespace nspc
{
   class Tech;
}
class nspc:: Tech
{
   public:
      void show()
      {
         cout<<"nspc::Tech::show()"<<endl;
      }
};
int main()
{
   nspc::Tech obj;
   obj.show();
   return 0;
}
```

Output:

```
nspc::Tech::show()
```

We can define methods as well outside the namespace.

```
#include<iostream>
using namespace std;
 namespace nspc
{
   class Tech
   {
   public:
      void show();
   };
}
void nspc:: Tech:: show()
{
        cout<<"nspc::Tech::show()"<<endl;
}
int main()
{
   nspc::Tech obj;
   obj.show();
   return 0;
}
```

Output:

```
nspc::Tech::show()
```

Namespaces in C++ - Cont'd

- **Nested Namespaces:**
  - Namespaces can be nested where we can define one namespace inside another name space as follows:

```
namespace namespace_name1
   {
      // code declarations
      namespace namespace_name2
      {
        // code declarations
      }
   }
```

•We can access members of nested namespace by using resolution operators as follows:

```
// to access members of namespace_name2
using namespace namespace_name1::namespace_name2;
// to access members of namespace_name1
using namespace namespace_name1;
```

Nested Namespaces - Example

```
#include<iostream>
using namespace std;
namespace firstSpace
{
   void func()
   {
      cout << "Inside firstSpace" << endl;
   }
  namespace secondSpace
   {
      void func()
      {
         cout << "Inside secondSpace" << endl;
      }
   }
}
using namespace firstSpace::secondSpace;
int main ()
{
   // This calls function from second name space.
    func();
    return 0;
}
```

Output:

```
Inside secondSpace
```

Namespaces in C++ - Cont'd

- **namespace extension :**
  - It is also possible to create more than one namespaces in the global space.
  - This can be done in two ways.
    - **namespaces having different names**
    - **Extending namespaces (Using same name twice)**

namespaces having different names - Example

```
#include <iostream>
using namespace std;
namespace first
{
  void func()
  {
      cout<<"func in first namespace\n";
   }
}
namespace second
{
  void func()
  {
      cout<<"func in second namespace\n";
   }
}
int main()
{
   first::func() ;
   second::func() ;
   return 0;
}
```

Output:

```
func in first namespace
func in second namespace
```

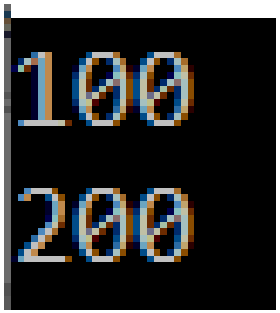
Namespaces in C++ - Cont'd

- **Extending namespaces (Using same name twice)**
  - It is also possible to create two namespace blocks having the same name.
  - The second namespace block is nothing but actually the continuation of the first namespace.
  - In simpler words, we can say that both the namespaces are not different but actually the same, which are being defined in parts.

Extending namespaces (Using same name twice) - Example

```
#include <iostream>
using namespace std;
namespace first
{
  int n1 = 100;
}
namespace  first
{
  int n2 = 200;
}
int main()
{
  cout << first::n1 <<"\n";
  cout << first::n2 <<"\n";
  return 0;
}
```

Output:



Namespaces in C++ - Cont'd

- **Unnamed Namespaces**
  - They are directly usable in the same program and are used for declaring unique identifiers.
  - In unnamed namespaces, name of the namespace is not mentioned in the declaration of namespace.
  - The name of the namespace is uniquely generated by the compiler.
  - **The unnamed namespaces we have created will only be accessible within the file we created it in.**
  - Unnamed namespaces are the replacement for the static declaration of variables.

Unnamed Namespaces - Example

```
#include <iostream>
using namespace std;
namespace // unnamed namespace
{
   void doSomething() // can only be accessed in this file
   {
      cout << "inside function do something\n";
   }
}
int main()
{
   doSomething(); // we can call doSomething() without a namespace prefix
   return 0;
}
```

Output:

```
inside function do something
```

```cpp
#include <iostream>
using namespace std;
namespace // unnamed namespace
{
   void doSomething() // can only be accessed in this file
   {
      cout << "inside function do something\n";
   }
}
int main()
{
   doSomething(); // we can call doSomething() without a namespace prefix
   return 0;
}
```

```cpp
#include <iostream>
using namespace std;
static void doSomething() // can only be accessed in this file
{
   cout << "inside function do something\n";
}
int main()
{
   doSomething(); // we can call doSomething() without a namespace prefix
   return 0;
}
```

Both are same

Output:

```
inside function do something
```

UNIT IV - TEMPLATES AND EXCEPTION HANDLING

- **Topics to be discussed,**
  - Function Template and Class Template
  - Namespaces
  - **Casting**
  - Exception Handling

Casting

- Casting is a conversion process wherein data can be changed from one type to another.
- C++ has two types of conversions:
  - **Implicit conversion:** Conversions are performed automatically by the compiler without the programmer's intervention.
  - **Example:**

```
int iVariable = 10;
float fVariable = iVariable; //Assigning an int to a float will trigger a conversion.
```

  - **Explicit conversion:** Conversions are performed only when explicitly specified by the programmer.
  - **Example:**

```
int iVariable = 20;
float fVariable = (float) iVariable / 10;
```

Casting – Cont'd

- The functionality of these explicit conversion operators is enough for most needs with fundamental data types.
- However, these operators can be applied indiscriminately on classes and pointers to classes, which can lead to code that while being syntactically correct can cause runtime errors.

```
// class type-casting
#include <iostream>
using namespace std;
class floatClass
{
   float i,j;
public:
   floatClass(float x,float y)
   {
      i=x;
      j=y;
   }
};
```

```
class intClass
{
   int x,y;
   public:
   intClass (int a, int b)
    {
      x=a;
      y=b;
    }
   int result()
   {
      return x+y;
   }
};
int main ()
{
  floatClass f(1.2,4.5);
  intClass  * pi;
  pi = (intClass*) &f;
  cout << pi->result();
  return 0;
}
```

Output:

```
-2144757350
```

- Traditional explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to.
- The subsequent call to member result will produce either a run-time error or a unexpected result.

Casting – Cont'd

- **In order to control these types of conversions between classes, we have four specific casting operators:**
  - dynamic_cast
  - reinterpret_cast
  - static_cast
  - const_cast.
- Their format is to follow the new type enclosed between angle-brackets (<>) and immediately after, the expression to be converted between parentheses.
  - dynamic_cast <new_type> (expression)
  - reinterpret_cast <new_type> (expression)
  - static_cast <new_type> (expression)
  - const_cast <new_type> (expression)
- The traditional type-casting equivalents to these expressions would be:
  - (new_type) expression
  - new_type (expression)

Casting – Cont'd

- **Syntax of the traditional explicit type casting:**

  (type) expression;

- For example, we have a floating pointing number 4.534, and to convert an integer value, then we write as:

```
int num;
num = (int) 4.534; // cast into int data type
cout << num;
```

- When the above statements are executed, the floating-point value will be cast into an integer data type using the cast () operator.
- And the float value is assigned to an integer num that truncates the decimal portion and displays only 4 as the integer value.

Casting – Cont'd

static_cast

- The static_cast operator is the most commonly used casting operator in C++.
- It performs compile-time type conversion and is mainly used for explicit conversions that are considered safe by the compiler.
- **Syntax :**

  static_cast <dest_type> (source);

  – The return value of static_cast will be of dest_type.
- The static_cast can be used to convert between related types, such as numeric types or pointers in the same inheritance hierarchy.

static_cast Example 1

```
#include <iostream>
using namespace std;
int main()
{
    float f = 3.5;
    cout << "\nThe Value of f: " << f;
     int i1 = f; // Implicit type cast
    cout << "\nThe Value of i1: " << i1;
     // using static_cast for float to int
    int i2 = static_cast<int>(f);
    cout << "\nThe Value of i2: " << i2;
}
```

Output:

```
The Value of f: 3.5
The Value of i1: 3
The Value of i2: 3
```

static_cast Example 2

```
#include <iostream>
#include <typeinfo>
using namespace std;
int main()
{
   int num = 10;
   // converting int to double
   double numDouble = static_cast<double>(num);

   // printing data type
   cout << typeid(num).name() << endl;

   // typecasting
   cout << typeid(static_cast<double>(num)).name() << endl;

   // printing double type t
   cout << typeid(numDouble).name() << endl;

   return 0;
}
```

Output:

```
i
d
d
```

typeid operator in C++

- It is used where the dynamic type or runtime type information of an object is needed.
- It is included in the <typeinfo> library.
- The typeid expression is an lvalue expression.
- **Syntax:**

  typeid(type);

  OR

  typeid(expression);
- **Parameters:** typeid operator accepts a parameter, based on the syntax used in the program:
  - ***type:*** This parameter is passed when the runtime type information of a variable or an object is needed. In this, there is no evaluation that needs to be done inside type and simply the type information is to be known.
  - ***expression:*** This parameter is passed when the runtime type information of an expression is needed. In this, the expression is first evaluated. Then the type information of the final result is then provided.

typeid operator - Example

```
#include <iostream>
#include <typeinfo>
using namespace std;
int main()
{
   int i1, i2;
   char c;
    // Get the type info using typeid operator
   const type_info& ti1 = typeid(i1);
   const type_info& ti2 = typeid(i2);
   const type_info& ti3 = typeid(c);
   if (ti1 == ti2)  // Check if both types are same
      cout << "i1 and i2 are of"
         << " similar type" << endl;
   else
      cout << "i1 and i2 are of"
         << " different type" << endl;
   if (ti2 == ti3)   // Check if both types are same
      cout << "i2 and c are of"
         << " similar type" << endl;
   else
      cout << "i2 and c are of"
         << " different type" << endl;
    return 0;
}
```

Output:

```
i1 and i2 are of similar type
i2 and c are of different type
```

static_cast Example 3

```
int main()
{
  int i = 25;
  char ch = 'a';
  int* iptr = (int*)&ch;
 cout<<*iptr;
   return 0;
}
```

Output:

```
1644040033
```

(Unexpected result)

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5    int i = 25;
6    char ch = 'a';
7    int* iptr = static_cast<int*>(&ch);
8    cout<<*iptr;
9    return 0;
10 }
```

d messages | CppCheck/Vera++ | CppCheck/Vera++ messages | Cscope

	Line	Message
		=== Build file: "no target" in "no project" (compiler: unkn
\CS320...		In function 'int main()':
\CS320...	7	error: invalid static_cast from type 'char*' to type 'int*'
		=== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0

static_cast for Inheritance in C++

```
#include <iostream>
using namespace std;
class Base
{
   public:
    void show()
   {
      cout<<"\nBase class show method";
   }
};
class Derived :public Base
{
   public:
   void show()
   {
      cout<<"\nDerived class show method";
   }
};
int main()
{
 Derived dobj;
 // Implicit cast allowed
 Base* bptr = (Base *)&dobj; //*bptr=&dobj;
 bptr->show();
 return 0;
}
```

Output:

```
Base class show method
```

```
#include <iostream>
using namespace std;
class Base
{
   public:
   void show()
   {
      cout<<"\nBase class show method";
   }
};
class Derived :public Base
{
   public:
   void show()
   {
      cout<<"\nDerived class show method";
   }
};
int main()
{
 Derived dobj;
 // upcasting using static_cast
 Base* bptr = static_cast<Base*>(&dobj);
 bptr->show();
 return 0;
}
```

Output:

```
Base class show method
```

In the previous example, we inherited the base class as public. What happens when we inherit it as private?

```
#include <iostream>
using namespace std;
class Base
{
   public:
    void show()
   {
      cout<<"\nBase class show method";
   }
};
class Derived :private Base
{
   public:
   void show()
   {
      cout<<"\nDerived class show method";
   }
};
int main()
{
 Derived dobj;
 // Implicit cast allowed
 Base* bptr = (Base *)&dobj; //*bptr=&dobj;
 bptr->show();
 return 0;
}
```

Output:

```
Base class show method
```

```
1  #include <iostream>
2  using namespace std;
3  class Base
4  {
5     public:
6     void show()
7     {
8        cout<<"\nBase class show method";
9     }
10 };
11 class Derived :private Base
12 {
13    public:
14    void show()
15    {
16       cout<<"\nDerived class show method";
17    }
18 };
19 int main()
20 {
21    Derived dobj;
22    // upcasting using static_cast
23    Base* bptr = static_cast<Base*>(&dobj);
24    bptr->show();
25    return 0;
26 }
```

```
                    === Build file: "no target" in "no project" (compiler:
F:\2024\CS320...    In function 'int main()':
F:\2024\CS320...  23  error: 'Base' is an inaccessible base of 'Derived'
                    === Build failed: 1 error(s), 0 warning(s) (0 minute(s),
```

Casting – Cont'd
dynamic_cast

- The dynamic_cast operator is mainly used to perform downcasting (converting a pointer / reference of a base class to a derived class).
- It ensures type safety by performing a runtime check to verify the validity of the conversion.
- **Syntax :**

  dynamic_cast <new_type> (expression);

  – If the conversion is not possible, dynamic_cast returns a null pointer (for pointer conversions) or throws a bad_cast exception (for reference conversions).

dynamic_cast – Example

```
#include <iostream>
using namespace std;
class Base
{
public:
  virtual void show()
  {
    cout << "Base class Show method" << endl;
  }
};
class Derived1 : public Base
{
public:
  virtual void show()
  {
    cout << "Derived1 class Show method" << endl;
  }
};
class Derived2 : public Base
{
public:
  virtual void show()
  {
    cout << "Derived2 class Show method" << endl;
  }
};
int main()
{
  Base* bptr = new Derived1();// base class pointer to derived class object
  // downcasting
  Derived1* d1ptr = dynamic_cast<Derived1*>(bptr);
  if (d1ptr) // checking if the typecasting is successful
  {
    d1ptr->show();
  }
  else
  {
    cout << "Failed to cast to Derived1" << endl;
  }
  // typecasting to other derived class
  Derived2* d2ptr = dynamic_cast<Derived2*>(bptr);
  if (d2ptr) // checking if the typecasting is successful
  {
    d2ptr->show();
  }
  else
  {
    cout << "Failed to cast to Derived2" << endl;
  }
  delete bptr;
  return 0;
}
```

Output:

```
Derived1 class Show method
Failed to cast to Derived2
```

- In this example, the first line of output is printed because the 'bptr' of the 'Base' type is successfully cast to the 'Derived1' type and show() function of the Derived1 class is invoked but the casting of the 'Base' type to 'Derived2' type is failed because 'bptr' points to a 'Derived1' object thus, the dynamic cast fails because the typecasting is not safe.

Casting – Cont'd

const_cast

- The const_cast operator is **used to modify the const or volatile qualifier of a variable**.
- It allows programmers to temporarily remove the constancy of an object and make modifications.
- Caution must be exercised when using const_cast, as modifying a const object can lead to undefined behavior.
- **Syntax :**

  const_cast <new_type> (expression);

const_cast - Example

```
#include <iostream>
using namespace std;
int main()
{
   const int num = 50;
   const int* cptr = &num;
   cout<<"\nold value is "<<*cptr<<"\n";
   int* ptr = const_cast<int*>(cptr);
   *ptr = 100;
   cout<<"new value is "<<*cptr;
   return 0;
}
```

Output:

```
old value is 50
new value is 100
```

Casting – Cont'd
reinterpret_cast

- The reinterpret_cast operator is used to **convert the pointer to any other type of pointer.**
- It does not perform any check whether the pointer converted is of the same type or not.
- **Syntax:**

  reinterpret_cast <new_type> (expression);
- Even if they are unrelated or incompatible, it enables us to convert a pointer of one type to a pointer of a different type.
- Because it might result in undeclared behaviour and system crashes if used carelessly, the reinterpret_cast operator is sometimes regarded as the most hazardous of the C++ type-casting operators.

reinterpret_cast - Example

```
#include <iostream>
using namespace std;
int main()
{
   int intData = 25;
   int* intPtr = &intData;
   // Attempting to reinterpret_cast int pointer to a float pointer
   float* floatPtr=reinterpret_cast<float*>(intPtr);
   // Accessing the original integer data through the float pointer
   int retrievedData = *reinterpret_cast<int*>(floatPtr);
   cout<< "Original Integer Data: " << intData<<endl;
   cout<< "Float Pointer Value: " <<*floatPtr << endl;
   cout<< "Retrieved Integer Data: " <<retrievedData << endl;
   cout << "Integer Address: " << intPtr << endl;
   cout << "Float Address: " << reinterpret_cast<void*>(floatPtr) << endl;
}
```

Output:

```
Original Integer Data: 25
Float Pointer Value: 3.50325e-44
Retrieved Integer Data: 25
Integer Address: 0x61fe08
Float Address: 0x61fe08
```

Casting – Cont'd

- ***Note:*** *const_cast and reinterpret_cast are generally not recommended as they vulnerable to different kinds of errors.*

UNIT IV - TEMPLATES AND EXCEPTION HANDLING

- **Topics to be discussed,**
  - Function Template and Class Template
  - Namespaces
  - Casting
  - **Exception Handling**

Exception Handling

- **What is Exception?**
  - The errors that occur at run-time are known as exceptions.
  - An exception is an unexpected problem that arises during the execution of a program our program terminates suddenly with some errors/issues.
  - **Types of C++ Exception**
    - There are two types of exceptions in C++
      - **Synchronous**
      - **Asynchronous**

Exception Handling – Cont'd

- **Synchronous:**
  - Exceptions that happen when something goes wrong because of a mistake in the input data or when the program is not equipped to handle the current type of data it's working with
  - For example, they occur due to different conditions such as division by zero, accessing an element out of bounds of an array, unable to open a file, running out of memory and many more.
- **Asynchronous**:
  - Exceptions that are beyond the program's control, such as disc failure, keyboard interrupts, etc.

Exception Handling – Cont'd

- **Exception Handling in C++ is a process to handle runtime errors.**
- If we don't handle the exception, it prints exception message and terminates the program.
- **The main objective of exception handling is to provide a way to detect and report the exception** condition so that necessary action can be taken without troubling the user.
- We perform exception handling so the normal flow of the application can be maintained even after runtime errors.
- In C++, exception handling is designed to handle only synchronized exceptions.
- In C++, exception is an event or object which is thrown at runtime.
- All exceptions are derived from **std::exception class**.

Exception - Example

```
#include<iostream>
using namespace std;
int main()
{
   int n1,n2;
   float res;
   char ch;
   while(true)
   {
      cout<<"\nEnter 2 numbers:";
      cin>>n1>>n2;
      res=n1/n2;
      cout<<"res="<<res;
      cout<<"\nDo you want to continue?(y/n)";
      cin>>ch;
      if(ch!='y')
         break;
   }
}
```

Output:

```
Enter 2 numbers:45 6
res=7
Do you want to continue?(y/n)y

Enter 2 numbers:23 2
res=11
Do you want to continue?(y/n)y

Enter 2 numbers:12 0

Process returned -1073741676 (0xC0000094)   execution time : 22.120 s
Press any key to continue.
```

Exception Handling – Cont'd

- **Exception Handling Mechanism**
  - Whenever an exception occurs in a C++ program, the **portion the program that detects the exception can inform that exception has occurred by throwing it**
  - On throwing an exception, the program control immediately stops the step by step execution of the code and jumps to the separate block of code known as an exception handler.
  - **The exception handler catches the exception and processes it without troubling the user**.
  - However, if there is no exception handler, the program terminates abnormally.
  - **C++ provides three constructs try, throw and catch, for implementing exception handling.**

Exception Handling – Cont'd

C++ try and catch

Syntax:

```
try
{
    // Code that might throw an exception
    throw SomeExceptionType("Error message");
}
catch( ExceptionName e1 )
{
    // catch block catches the exception that is thrown from try block
}
```

- **try**
  - The try keyword represents a block of code that may throw an exception placed inside the try block.
  - It's followed by one or more catch blocks.
  - If an exception occurs, try block throws that exception.

Exception Handling – Cont'd

- **catch**
  - The catch statement represents a block of code that is executed when a particular exception is thrown from the try block.
  - The code to handle the exception is written inside the catch block.
- **throw**
  - An exception in C++ can be thrown using the throw keyword.
  - When a program encounters a throw statement, then it immediately terminates the current function and starts finding a matching catch block to handle the thrown exception.

Exception Handling – Example 1

```
#include <iostream>
using namespace std;
int main()
{
   int x = -1;
   cout << "Before try \n";
   try
   {
      cout << "Inside try \n";
      if (x < 0)
      {
         throw x;
         cout << "After throw (Never executed) \n";
      }
   }
   catch (int x)
   {
        cout << "Exception Caught \n";
   }
   cout << "After Caught (Will be executed) \n";
   return 0;
}
```

Output:

```
Before try
Inside try
Exception Caught
After Caught (Will be executed)
```

Exception Handling – Example 2

```
#include<iostream>
using namespace std;
int main()
{
   int n1,n2;
   float res;
   char ch;
   while(true)
   {
      cout<<"\nEnter 2 numbers:";
      cin>>n1>>n2;
      try
      {
         if (n2==0)
            throw 0;
         res=static_cast<float>(n1)/n2;
         cout<<"res="<<res;
         cout<<"\nDo you want to continue?(y/n)";
         cin>>ch;
         if(ch!='y')
            break;
      }
      catch(int exp)
      {
         cout<<"Error:cannot divide by "<<exp;
      }
   }
}
```

Output:

```
Enter 2 numbers:23 4
res=5.75
Do you want to continue?(y/n)y

Enter 2 numbers:4 0
Error:cannot divide by 0
Enter 2 numbers:34 5
res=6.8
Do you want to continue?(y/n)n

Process returned 0 (0x0)   execution time : 30.233 s
Press any key to continue.
```

Exception Handling – Cont'd

```
try
 {
   // code
}
catch (exception1)
{
   // code
}
catch (exception2)
{
   // code
}
```

- **Multiple catch Statements**
  - In C++, we can use multiple catch statements for different kinds of exceptions that can result from a single block of code.

Multiple catch Statements – Example 1

```
#include <stdexcept>
using namespace std;
int x = 5;
int main()
{
   try
   {
      if (x == 0)
         throw x;
      else if (x > 0)
         throw 'x';
      else
         throw "x is negative";
   }
   catch (int i)
   {
         cout << "Caught an int exception: " << i << endl;
   }
   catch (char c)
   {
         cout << "Caught a char exception: " << c << endl;
   }
   catch (char* str)
   {
         cout << "Caught a string exception: " << str << endl;
   }
}
```

Output:

```
Caught a char exception: x
```

Exception Handling – Cont'd

```
try
{
    // code
}
catch (...)
{
    // code
}
```

- **Catching All Types of Exceptions**
  - In exception handling, it is important that we know the types of exceptions that can occur due to the code in our try statement.
  - This is so that we can use the appropriate catch parameters.
  - Otherwise, the try...catch statements might not work properly.
  - If we do not know the types of exceptions that can occur in our try block, then we can use the ellipsis symbol ... as our catch parameter.

Exception Handling – Cont'd

```
try
{
   // code
}
catch (exception1)
 {
   // code
}
catch (exception2)
 {
   // code
}
catch (...)
{
   // code
}
```

- Our program catches exception1 if that exception occurs.
- If not, it will catch exception2 if it occurs.
- If there is an error that is neither exception1 nor exception2, then the code inside of catch (...) {} is executed.
- **Note:**
- catch (...) {} should always be the final block in our try...catch statement.
- This is because this block catches all possible exceptions and acts as the default catch block
- It is not compulsory to include the default catch block in our code.

Multiple catch Statements - Example

```
#include<iostream>
using namespace std;
int main()
{
   int ind1,ind2;
   int arr[5]={45,34,78,0,22};
   float res;
   char ch;
   while(true)
   {
      cout<<"\nEnter 2 index numbers:";
      cin>>ind1>>ind2;
      try
      {
         if (ind1>4 || ind2>4)
            throw "Error:Array index out of bounds";
         if(arr[ind2]==0)
            throw 0;
        res=static_cast<float>(arr[ind1])/arr[ind2];
        cout<<"res="<<res;
```

```
        cout<<"\nDo you want to continue?(y/n)";
        cin>>ch;
         if(ch!='y')
            break;
    }
    catch(const char* emsg)
    {
        cout<<emsg;
    }
    catch(int exp)
    {
        cout<<"Error:cannot divide by "<<exp;
    }
    catch (...)
    {
        cout << "Unexpected exception!" << endl;
    }
  }
}
```

Output:

```
Enter 2 index numbers:2 0
res=1.73333
Do you want to continue?(y/n)y

Enter 2 index numbers:1 3
Error:cannot divide by 0
Enter 2 index numbers:4 1
res=0.647059
Do you want to continue?(y/n)y

Enter 2 index numbers:1 5
Error:Array index out of bounds
Enter 2 index numbers:1 4
res=1.54545
Do you want to continue?(y/n)n

Process returned 0 (0x0)   execution time : 79.818 s
Press any key to continue.
```

Exception Handling – Cont'd

Throwing Exceptions from C++ constructors

- An exception should be thrown from a C++ constructor whenever an object cannot be properly constructed or initialized.
- Since there is no way to recover from failed object construction, an exception should be thrown in such cases.
- Since C++ constructors do not have a return type, it is not possible to use return codes.
- Therefore, the best practice is for constructors to throw an exception to signal failure.
- The throw statement can be used to throw a C++ exception and exit the constructor code.

Throwing Exceptions from C++ constructors - Example

```
#include <iostream>
using namespace std;
class Rectangle
{
  private:
    int length;
    int breadth;
  public:
    Rectangle(int l, int b)
    {
      if (l < 0 || b < 0)
      {
        throw 1;
      }
      else
      {
        length = l;
        breadth = b;
      }
    }
    void Display()
    {
      cout << "Length: " << length << " Breadth: " << breadth;
    }
};
```

```
int main()
{
  try
  {
    Rectangle r1(10, -5);
    r1.Display();
  }
  catch (int num)
  {
    cout << "Rectangle Object Creation Failed";
  }
}
```

Output:

```
Rectangle Object Creation Failed
```

Exception Handling – Cont'd

- Implicit type conversion doesn't happen for primitive types.

```
#include <iostream>
using namespace std;
int main()
{
   try
   {
      throw 'a';
   }
   catch (int x)
   {
      cout << "Caught " << x;
   }
   catch (...)
   {
      cout << "Default Exception\n";
   }
   return 0;
}
```

Output:

```
Default Exception
```

Exception Handling – Cont'd

- If an exception is thrown and not caught anywhere, the program terminates abnormally.

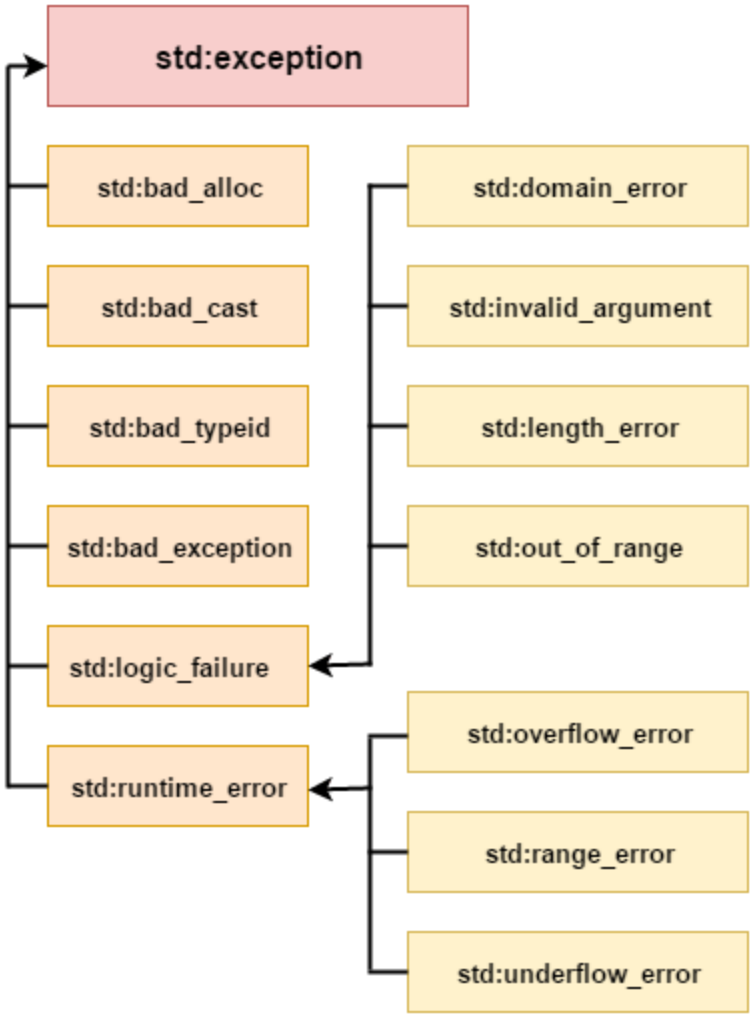
```
#include <iostream>
using namespace std;
int main()
{
   try
   {
      throw 'a';
   }
   catch (int x)
   {
      cout << "Exception Caught ";
   }
   return 0;
}
```

Output:

```
terminate called after throwing an instance of 'char'
```

Exception Handling – Cont'd

C++ Standard Exception



- In C++ standard exceptions are defined in <exception> class that we can use inside our programs.

Exception Handling – Cont'd

C++ Standard Exceptions

- **std::exception** - Parent class of all the standard C++ exceptions.
- **logic_error** - Exception happens in the internal logical of a program.
  - **domain_error** - Exception due to use of invalid domain.
  - **invalid argument** - Exception due to invalid argument.
  - **out_of_range** - Exception due to out of range i.e. size requirement exceeds allocation.
  - **length_error** - Exception due to length error.

Exception Handling – Cont'd

C++ Standard Exceptions

- **runtime_error** - Exception happens during runtime.
  - **range_error** - Exception due to range errors in internal computations.
  - **overflow_error** - Exception due to arithmetic overflow errors.
  - **underflow_error** - Exception due to arithmetic underflow errors
- **bad_alloc** - Exception happens when memory allocation with new() fails.
- **bad_cast** - Exception happens when dynamic cast fails.
- **bad_exception** - Exception is specially designed to be listed in the dynamic-exception-specifier.
- **bad_typeid** - Exception thrown by typeid.

Standard Exception Example 1

```
#include <iostream>
using namespace std;
int main()
{
   try
   {
      int num1, num2;
      cout << "Enter two numbers: ";
      cin >> num1 >> num2;
      if (num2 == 0)
      {
         throw runtime_error("Divide by zero exception");
      }
      int result = num1 / num2;
      cout << "Result: " << result <<endl;
   }
   catch (const exception& e)
   {
      cout << "Exception caught: " << e.what() << std::endl;
   }
   return 0;
}
```

Output 1:

```
Enter two numbers: 24 2
Result: 12
```

Output 2:

```
Enter two numbers: 23 0
Exception caught: Divide by zero exception
```

Standard Exception Example 2

```
#include<iostream>
#include <stdexcept>
using namespace std;
int divide(int a, int b)
{
   if (b == 0)
   {
      throw invalid_argument("division by zero");
   }
   return a / b;
}
int main()
{
   try
   {
      int result = divide(1, 0);
      cout << result << endl;
   }
   catch (const invalid_argument& e)
   {
      cout << "An exception occurred: " << e.what() << endl;
   }
   return 0;
}
```

Output:

```
An exception occurred: division by zero
```

Exception Handling – Cont'd
re-throwing an Exception

- Re-throwing an exception in C++ involves catching an exception within a try block and **instead of dealing with it locally, throwing it again to be caught by an outer catch block.**
- By doing this, we preserve the type and details of the exception ensuring that it can be handled at the appropriate level within our program.
- This approach becomes particularly valuable when managing exceptions at multiple levels or when additional actions need to be performed before resolving the exception.

re-throwing an Exception - Example

```
#include <iostream>
using namespace std;
void division(int n1,int n2)
{
   try
   {
      if(n2==0)
         throw n2;
      else
         cout<<"n1/n2="<<(float)n1/n2;
   }
   catch(int)
   {
      cout<<"\nCaught an exception as first throwing";
      throw;
   }
}
```

```
int main()
{
   int a,b;
   cout<<"\nEnter 2 numbers:";
   cin>>a>>b;
   try
   {
      division(a,b);
   }
   catch(int)
   {
      cout<<"\nCaught an exception as re-throwing";
   }
   return 0;
}
```

Output 1:

```
Enter 2 numbers:45 6
n1/n2=7.5
```

Output 2:

```
Enter 2 numbers:23 0

Caught an exception as first throwing
Caught an exception as re-throwing
```

Exception Handling – Cont'd

- In C++, **try/catch blocks can be nested**.
- Also, an exception can be re-thrown using "throw; ".

```
#include <iostream>
using namespace std;
int main()
{
   // nesting of try/catch
   try {
          try
          {
             throw 20;
          }
          catch (int n)
          {
             cout << "Handle Partially\n";
             throw; // Re-throwing an exception
          }
     }
   catch (int n)
   {
      cout << "Handle remaining\n ";
   }
   return 0;
}
```

Output:

```
Handle Partially
Handle remaining
```

Exception Handling – Cont'd

- When an exception is thrown, all objects created inside the enclosing try block are destroyed before the control is transferred to the catch block.

```
#include <iostream>
using namespace std;
class Demo
{
public:
      Demo()
      {
         cout << "Constructor of Demo " << endl;
      }
      ~Demo()
      {
         cout << "Destructor of Demo " << endl;
      }
};
int main()
{
   try
   {
      Demo obj;
      throw 10;
   }
   catch (int i)
   {
        cout << "Caught " << i << endl;
   }
}
```

Output:

```
Constructor of Demo
Destructor of Demo
Caught 10
```

Exception Handling – Cont'd

User-Defined Exceptions

- The C++ **std::exception class allows us to define objects that can be thrown as exceptions**.
- This class has been defined in the <exception> header.
- The class provides us with a virtual member function named what.
- This function returns a null-terminated character sequence of type char *.
- We can overwrite it in derived classes to have an exception description.

User-Defined Exceptions - Example

```
#include<iostream>
using namespace std;
#include <exception>
class MyException:public exception
{
   public:
      char *what()
      {
         return "My Custom Exception";
      }
};
int Division(int a, int b)
{
   if (b == 0)
      throw MyException ();
   return a / b;
}
```

```
int main()
{
   int x = 10, y = 0, z;
   try
   {
      z = Division (x, y);
      cout << z << endl;
   }
   catch (MyException ME)
   {
      cout << "Division By Zero" << endl;
      cout << ME.what () << endl;;
   }
   cout << "End of the Program" << endl;
}
```

Output:

```
Division By Zero
My Custom Exception
End of the Program
```

Exception Handling – Cont'd

- **How to make the function throws something in C++?**
  - when a function is throwing, we can declare that this function throws something.

For example,

```
int Division(int a, int b) throw (MyException)
{
        if (b == 0)
            throw MyException();
        return a / b;
}
```

- This Division function declares that it throws some exception i.e. MyException.
- This is optional in C++.
- Whether we want to write or not is up to us.

Exception Handling – Cont'd

- So, whatever the type of value we are throwing, we can mention that in the brackets
- And if there are more values then we can mention them with commas

```
int Division(int a, int b) throw (int)
{
      if (b == 0)
            throw 1;
      return a / b;
}
```

```
int Division(int a, int b) throw (int, MyException)
{
    if (b == 0)
        throw 1;
    if (b == 1)
        throw MyException();
    return a / b;
}
```

function throws something - Example

```
#include<iostream>
using namespace std;
#include <exception>
class MyException:public exception
{
   public:
   char * what()
   {
      return "My Custom Exception";
   }
};
int Division(int a, int b) throw (int, MyException)
{
   if (b == 0)
      throw 1;
   if (b == 1)
      throw MyException();
   return a / b;
}
```

```
int main()
{
   int x = 10, y = 1, z;
   try
   {
      z = Division (x, y);
      cout << z << endl;
   }
   catch (int x)
   {
      cout << "Division By Zero Error" << endl;
   }
   catch (MyException ME)
   {
      cout << "Division By One Error" << endl;
      cout << ME.what () << endl;
   }
   cout << "End of the Program" << endl;
}
```

Output:

```
Division By One Error
My Custom Exception
End of the Program
```