CSE 1325

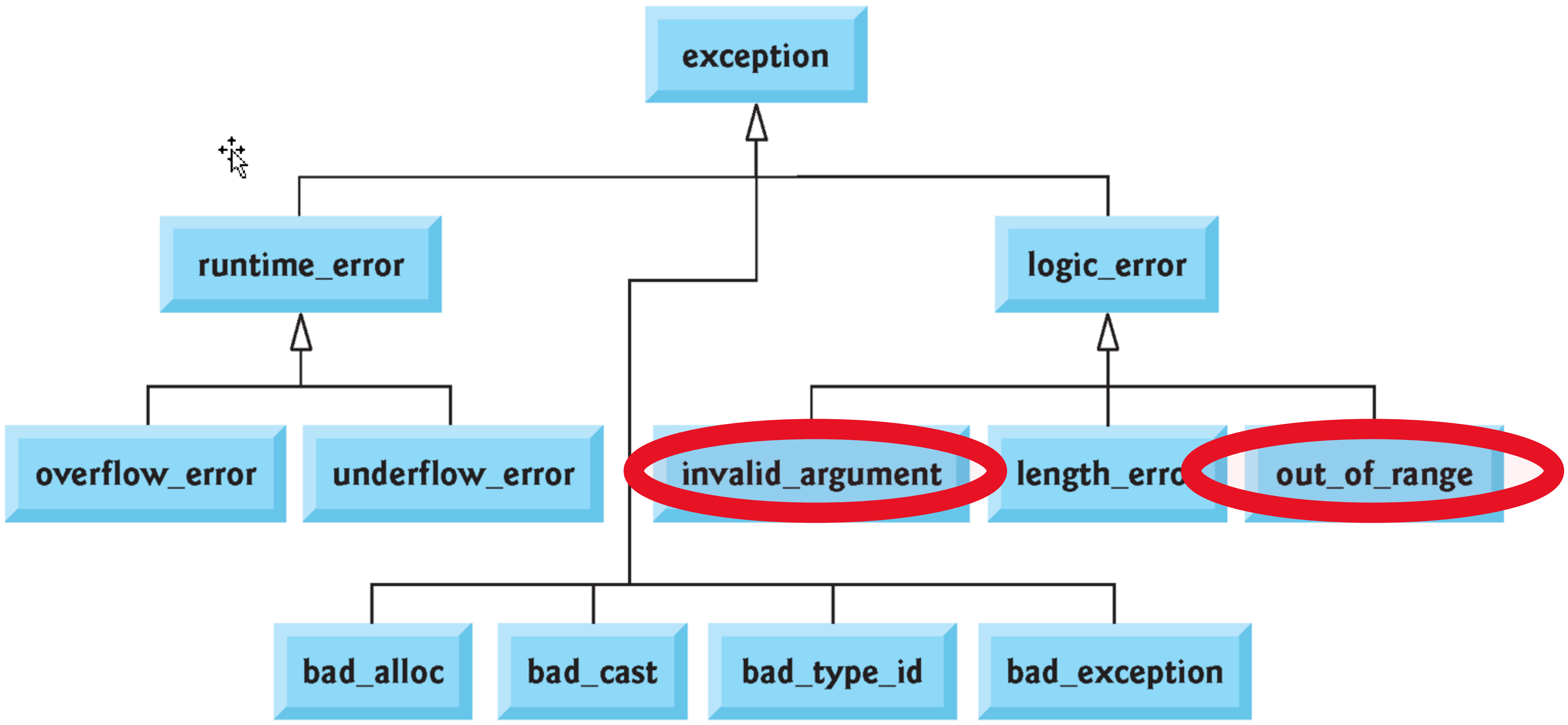
Week of 11/30/2020

Instructor : Donna French

C++ Standard Exceptions

- Experience has shown that exceptions fall nicely into a number of categories.

- The C++ Standard Library includes a hierarchy of exception classes.


exception
runtime_error
logic_error
overflow_error
underflow_error
invalid_argument
length_error
out_of_range
bad_alloc
bad_cast
bad_type_id
bad_exception

C++ Standard Exceptions

Exceptions thrown by C++ operators

- `bad_alloc`
  - thrown by `new`
- `bad_cast`
  - thrown by `dynamic_cast`
- `bad_typeid`
  - thrown by `typeid`

C++ Standard Exceptions

Exceptions indicating errors in program logic

- `invalid_argument`
  - indicates that a function received an invalid argument
- `length_error`
  - indicates that a length larger than the maximum size allowed for the object being manipulated was used for that object
- `out_of_range`
  - indicates that a value, such as a subscript into an array, exceeded its allowed range of values

C++ Standard Exceptions

Exceptions indicating errors during run time

- `overflow_error`
  - describes an arithmetic overflow error
    - the result of an arithmetic operation is larger than the largest number that can be stored in the computer
- `underflow_error`
  - describes an arithmetic underflow error
    - the result of an arithmetic operation is smaller than the smallest number that can be stored in the computer

C++ Standard Exceptions

- This hierarchy is headed by base-class `exception` (defined in header file `<exception>`), which contains `virtual` function `what` that derived classes can override to issue appropriate error messages.

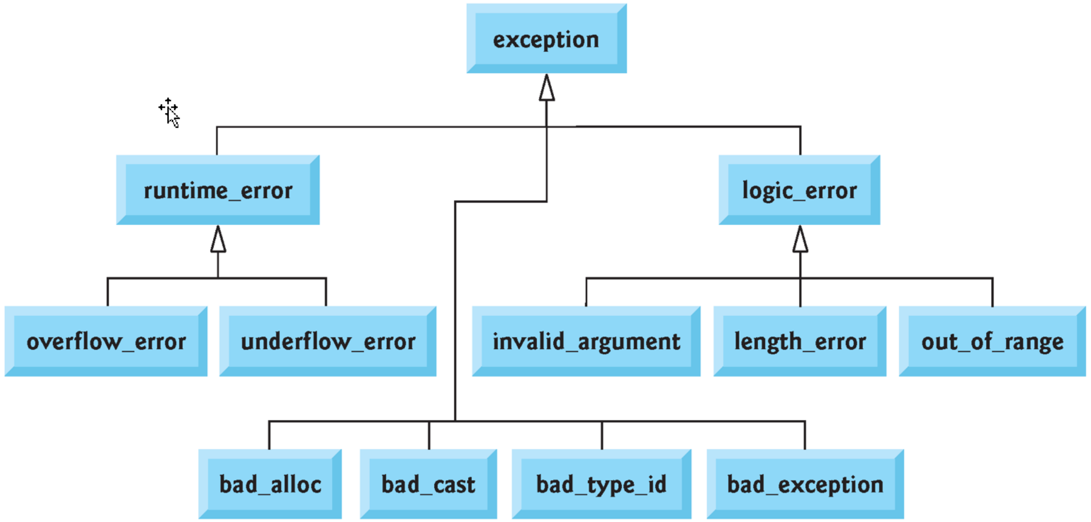
- If a catch handler catches a reference to an exception of a base-class type, it also can catch a reference to all objects of classes derived publicly from that base class—this allows for polymorphic processing of related errors

```
try
{
  switch (Choice)
  {
    case 1 :
      throw runtime_error("Threw runtime_error");
      break;
    case 2 :
      throw overflow_error ("Threw overflow error");
      break;
    case 3 :
      throw underflow_error ("Threw underflow error");
      break;
    case 4 :
      throw bad_alloc();
      break;
    case 5 :
      throw bad_cast("x");
      break;
    case 6 :
      aptr = nullptr;
      cout << typeid(*aptr).name() << endl;
      throw bad_typeid();
      break;
```


```
    case 6 :
      aptr = nullptr;
      cout << typeid(*aptr).name() << endl;
      throw bad_typeid();
      break;
```


```
    case 7 :
      throw bad_exception();
      break;
    case 8 :
      throw logic_error("Threw logic error");
      break;
    case 9 :
      throw invalid_argument("Threw invalid_argument");
      break;
    case 10 :
      throw length_error("Threw length_error");
      break;
    case 11 :
      throw out_of_range("Threw out_of_range");
      break;
    default :
      cout << "Don't know what to throw" << endl;
  }
}
```

```
try
{
     switch...
}

catch (runtime_error &say)
{
      cout << say.what() << endl;
}
catch (logic_error &say)
{
      cout << say.what() << endl;
}
catch (exception &say)
{
      cout << say.what() << endl;
}
```

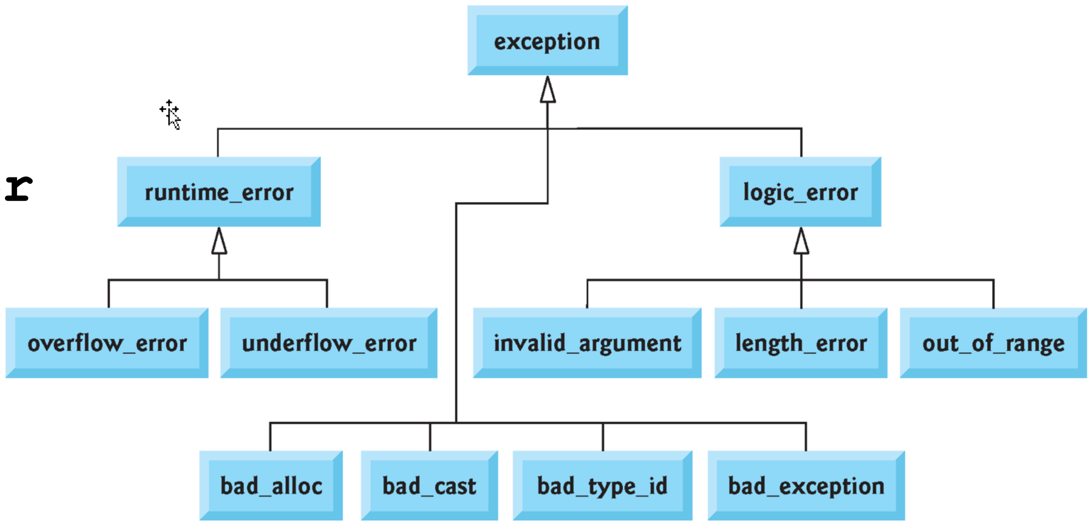



```
 0. Exit
 1. runtime_error
 2. overflow_error
 3. underflow_error
 4. bad_alloc
 5. bad_cast
 6. bad_typeid
 7. bad_exception
 8. logic_error
 9. invalid_argument
10. length_error
11. out_of_range

Enter Choice
```

0. Exit
1. runtime_error
2. overflow_error
3. underflow_error
4. bad_alloc
5. bad_cast
6. bad_typeid
7. bad_exception
8. logic_error
9. invalid_argument
10. length_error
11. out_of_range

Enter Choice 1
Threw runtime_error

```
catch (runtime_error &say)
{
   cout << say.what() << endl;
}
catch (logic_error &say)
{
   cout << say.what() << endl;
}
catch (exception &say)
{
   cout << say.what() << endl;
}
```

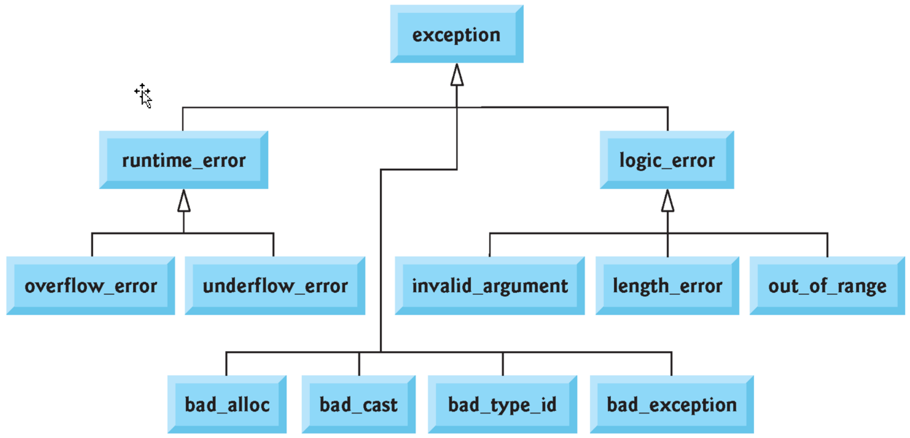


0. Exit
1. runtime_error
2. overflow_error
3. underflow_error
4. bad_alloc
5. bad_cast
6. bad_typeid
7. bad_exception
8. logic_error
9. invalid_argument
10. length_error
11. out_of_range

Enter Choice 2
Threw overflow error

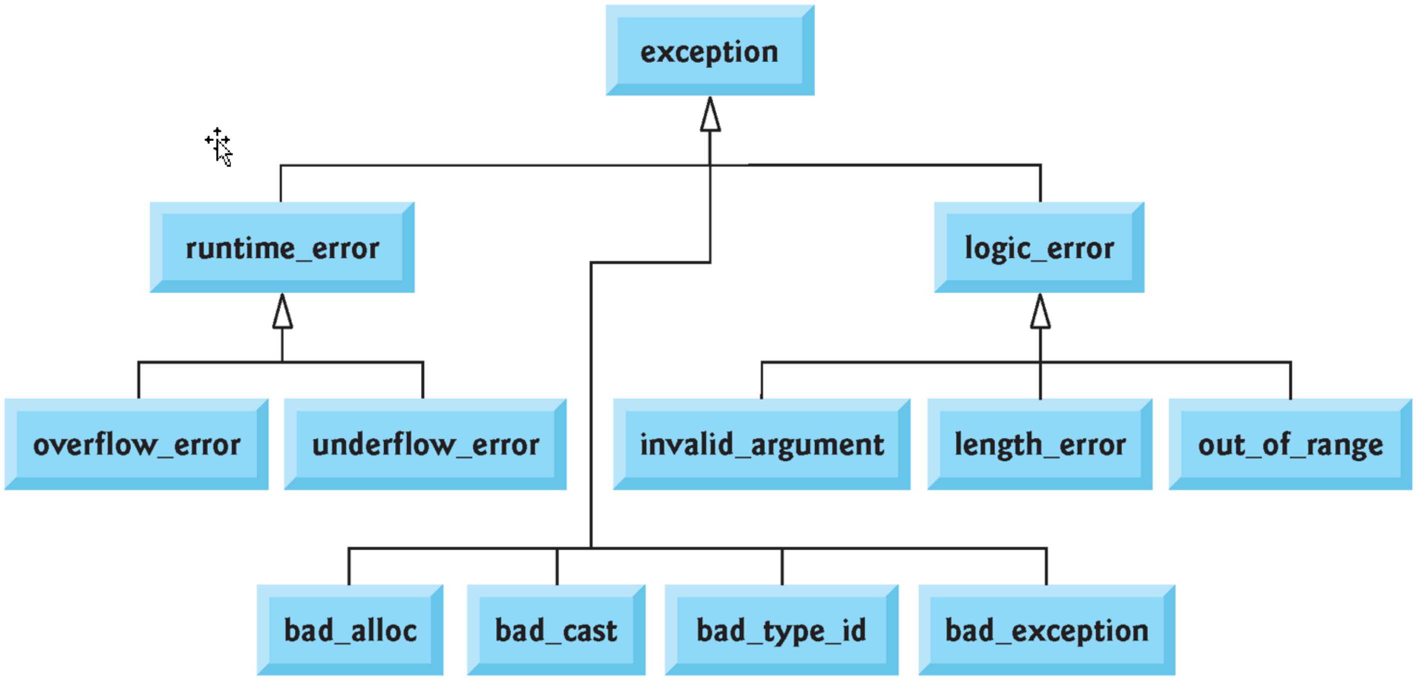
```
 0. Exit
 1. runtime_error
 2. overflow_error
 3. underflow_error
 4. bad_alloc
 5. bad_cast
 6. bad_typeid
 7. bad_exception
 8. logic_error
 9. invalid_argument
10. length_error
11. out_of_range

Enter Choice 3
Threw underflow error
```

```
 0. Exit
 1. runtime_error
 2. overflow_error
 3. underflow_error
 4. bad_alloc
 5. bad_cast
 6. bad_typeid
 7. bad_exception
 8. logic_error
 9. invalid_argument
10. length_error
11. out_of_range
Enter Choice 4
std::bad_alloc
```



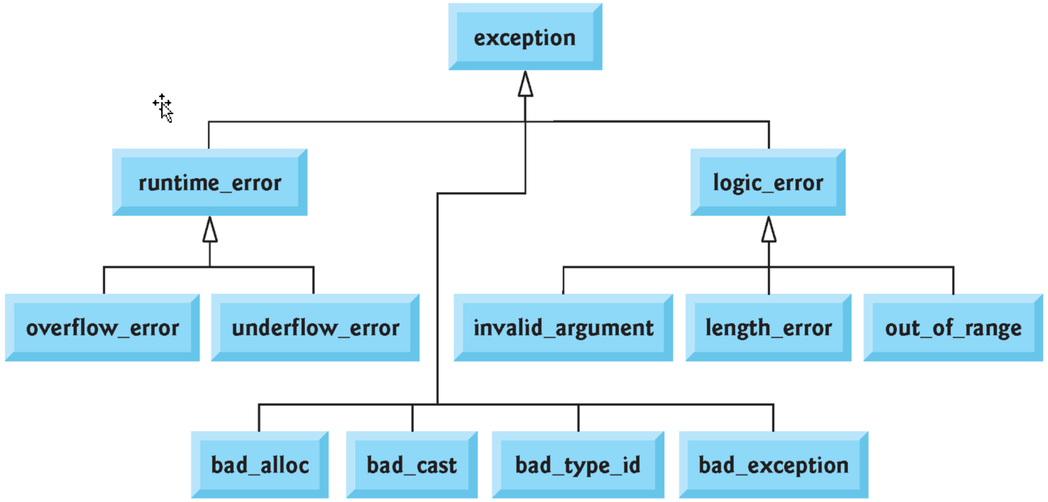

```
catch (runtime_error &say)
{
    cout << say.what() << endl;
}
catch (logic_error &say)
{
    cout << say.what() << endl;
}
catch (exception &say)
{
    cout << say.what() << endl;
}
```

```
student@cse1325:/media/sf_VM/C
Exiting program -

Missing command line parameter
CANDYFILENAME

throw invalid_argument("\n\nMi
TOTFILENAME HOUSEFILENAME CAND
```




```
try
{
   get_command_line_params(argc, argv, TOTFN, HFN, CFN);
}
catch (invalid_argument& say)
{
   cout << "Exiting program - " << say.what() << endl;
   exit(0);
}
```

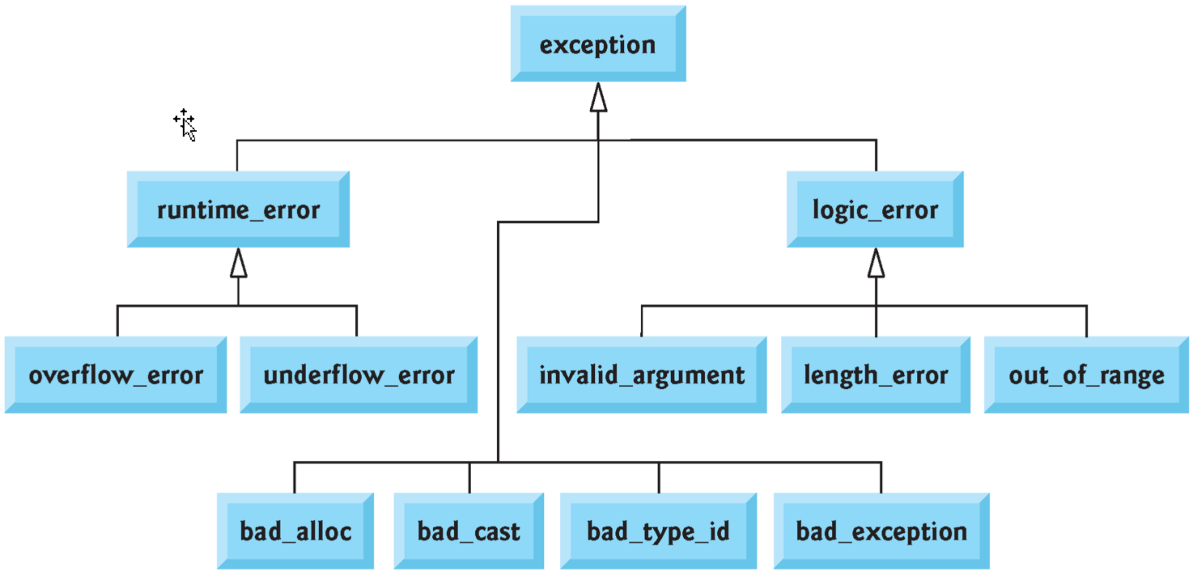
```
catch (invalid_argument& say)
{
  cout << "Exiting program - " << s
  exit(0);
}

catch (logic_error& say)
{
  cout << "Exiting program - " << say.what() << endl;
  exit(0);
}

catch (exception& say)
{
  cout << "Exiting program - " << say.what() << endl;
  exit(0);
}
```



Custom Exception Handling

Class `runtime_error` and `logic_error` are derived class of `exception` (from header file <`exception`>)

Class `exception` is the standard C++ base class for exceptions in the C++ Standard Library.



Custom Exception Handling

A typical exception class that derives from the `runtime_error` class defines only a constructor that passes an error-message string to the base-class `runtime_error` constructor.

Every exception class that derives directly or indirectly from `exception` contains the `virtual` function `what()`, which returns an exception object's error message.

You are not required to derive a custom exception class from the standard exception classes provided by C++ but doing so allows you to use the `virtual` function `what()` to obtain an appropriate error message and also allows you to polymorphically process the exceptions by catching a reference to the base-class type.

```
student@cse1325:/media/sf_VM$ ./custexFruitDemo.e
Enter any fruit except apple or orange
You followed instructions and entered pear

student@cse1325:/media/sf_VM$ ./custexFruitDemo.e
Enter any fruit except apple or orange
How dare you enter apple!!

student@cse1325:/media/sf_VM$ ./custexFruitDemo.e
Enter any fruit except apple or orange
Ewwww...orange?! - really?

student@cse1325:/media/sf_VM$
```

```
int main(void)
{
   string fruit;

   try
   {
      getFruit(fruit);
      cout << "You followed instructions and entered " << fruit << endl;
   }
```

```cpp
void getFruit(string &fruit)
{
  cout << "Enter any fruit except apple or orange ";
  cin >> fruit;

  if (fruit == "apple")
  {
    throw AppleEx();
  }
  if (fruit == "orange")
  {
    throw OrangeEx();
  }
}
```

```cpp
int main(void)
{
   string fruit;

   try
   {
      getFruit(fruit);
      cout << "You followed instructions and entered "
            << fruit << endl;
   }
```

```
int main(void)
{
   string fruit;

   try
   {
      getFruit(fruit);
      cout << "You followed instructions and entered " << fruit << endl;
   }
   catch (const AppleEx &say)
   {
      cout << say.what();
   }
   catch(const OrangeEx &say)
   {
      cout << say.what();
   }

   return 0;
}
```

```
class AppleEx : public std::logic_error
{
  public:
    AppleEx() : std::logic_error{"How dare you enter apple!!\n"}
    {
    }
};

class OrangeEx : public std::logic_error
{
  public:
    OrangeEx() : std::logic_error{"Ewwww...orange?! - really?\n"}
    {
    }
};
```

Both classes publicly inherit from `logic_error`; therefore, inherit the ability to store a statement in the object which can later be retrieved using `what()`.

Casting

What if we need a member function that we don't want to inherit from the base class?

What if we add a member function to a derived class?

```
class Circle : public Shape
{
	public:
		float getDiameter()
		{
			return 2*(dim1 + dim2);
		}
};
```

If this was defined in `Shape`, then `Square`, `Rectangle` and `Triangle` would inherit it for no reason.

Casting

```
for (auto it : MyShapes)
{
   it->Hello();
   cout << it->getName() << "'s area is " << it->getarea() << endl;
   cout << it->getDiameter() << endl;
}
```

```
student@cse1325:/media/sf_VM$ g++ Shape5.cpp -g -std=c++11
Shape5.cpp: In function 'int main()':
Shape5.cpp:155:15: error: 'class Shape' has no member named 'getDiameter'
   cout << it->getDiameter() << endl;
               ^
```

Dynamic Casting

```
for (auto it : MyShapes)
{
      it->Hello();
      cout << it->getName() << "'s area is " << it->getarea() << endl;

      Circle* IamaCircle = dynamic_cast<Circle*>(it);
      if (IamaCircle != nullptr)
      {
         cout << "My diameter is " << IamaCircle->getDiameter() << endl;
      }
}
```

`dynamic_cast` returns a value of `nullptr` if the input object is not of the requested type.

In this `for` loop, `dynamic_cast` will not be equal to `nullptr` when `it` is a `Circle`.

`IamaCircle` is then a pointer to `Circle` that can call `Circle`'s member function `getDiameter()`.

Dynamic Casting

```
for (auto it : MyShapes)
{
      it->Hello();
      cout << it->getName() << "'s area is " << it->getarea() << endl;

      Circle* IamaCircle = dynamic_cast<Circle*>(it);
      if (IamaCircle != nullptr)
      {
         cout << "My diameter is " << it->getDiameter() << endl;
      }
}
```

```
Shape5.cpp: In function 'int main()':
Shape5.cpp:164:37: error: 'class Shape' has no member named 'getDiameter'
    cout << "My diameter is " << it->getDiameter() << endl;
                                     ^
```

Static Casting

```
Square S2("S2", 5);

cout << S2.getName() << " is a " << S2.MySquareFunction() << endl;
```

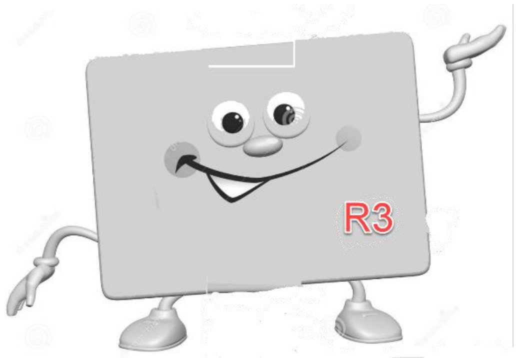
```
S2 is a Square
```

```
class Square : public Rectangle
{
   public:
      string MySquareFunction(void)
      {
         return "Square";
      }
```

Static Casting


```
Square S2("S2", 5);
cout << S2.getName() << " is a " << S2.MySquareFunction() << er
```

```
Rectangle R3 = static_cast<Rectangle>(S2);
```




```
cout << S2.getName() << " is a " << S2.MySquareFunction() <<
```

S2 is a Square

```
cout << R3.getName() << " is a " << R3.MySquareFunction() << endl;
```

```
Shape5.cpp: In function 'int main()':
Shape5.cpp:173:41: error: 'class Rectangle' has no member named 'MySquareFunctio
n'
  cout << R3.getName() << " is a " << R3.MySquareFunction() << endl;
                                          ^
```

Static Casting

```
Square S2("S2", 5);
cout << S2.getName() << " is a " << S2.MySquareFunction() << endl;
Rectangle R3 = static_cast<Rectangle>(S2);

Square S3 = static_cast<Square>(R3);
```

Can we cast `R3` back to a `Square` since it was a `Square` before being cast to a `Rectangle`?

```
Shape5.cpp: In function 'int main()':
Shape5.cpp:174:36: error: no matching function for call to 'Square::Square(Recta
ngle&)'
  Square S3 = static_cast<Square>(R3);
                                    ^

Shape5.cpp:98:3: note: candidate: Square::Square(std::__cxx11::string, float)
   Square(string shapeName, float size) : Rectangle()
   ^

Shape5.cpp:98:3: note:   candidate expects 2 arguments, 1 provided
Shape5.cpp:95:7: note: candidate: Square::Square(const Square&)
 class Square : public Rectangle
       ^

Shape5.cpp:95:7: note:   no known conversion for argument 1 from 'Rectangle' to
'const Square&'
```

Static Casting

```
Square S2("S2", 5);
cout << S2.getName() << " is a " << S2.MySquareFunction() << endl;
Rectangle R3 = static_cast<Rectangle>(S2);

Square S3 = static_cast<Square>(R3);
```

`Rectangle` is the base class for `Square`.

Derived class `Square` can be cast to its base class `Rectangle` but base class `Rectangle` cannot be cast to derived class `Square`.

Any extra/added after inheritance properties of the derived class are lost when cast to the base class.

A base class object cannot be cast to a derived class object because then it would be an object of that type without any of the extra/added after inheritance properties that other objects of that derived class have.

Function templates enable you to conveniently specify a variety of related (overloaded) functions—called function-template specializations.

Class templates enable you to conveniently specify a variety of related classes—called class-template specializations.

Programming with templates is known as generic programming.

Function templates and class templates are like stencils out of which we trace shapes; function-template specializations and class-template specializations are like the separate tracings that all have the same shape, but could, for example, be drawn in different colors , line thickneses and textures.

Class Templates

Class templates encourage software reusability by enabling a variety of type-specific class-template specializations to be instantiated from a single class template.

Class templates are called parameterized types, because they require one or more type parameters to specify how to customize a generic class template to form a class-template specialization.

When a particular specialization is needed, you use a concise, simple notation, and the compiler writes the specialization source code.

Class Templates

To create a template specialization with a user-defined type, the user-defined type must meet the template's requirements.

For example, the template might compare objects of the user-define type with < to determine sorting order. Or, the template might call a specific member function on an object of the user-defined type.

If the user-defined type does not overload the required operator or provide the required functions, compilation errors occur.

Class Templates

So what if we want to create a class that contains the data and member functions to emulate stack behavior?

We will use a vector but want to be able to call functions like push and pop.

And, we want to be able to use our stack with `int`s, `char`s and `double`s.

```
#include <iostream>
#include "Stack.h"

using namespace std;

int main()
{
     Stack<double> doubleStack; // create a Stack of double
     Stack<int> intStack;       // create a Stack of int
     Stack<char> charStack;     // create a Stack of int
```

This is instantiating objects `doubleStack`, `intStack` and `charStack`.

They are using template class `Stack`. The part inside the <> is the type we want `Stack` to substitute in the template.

```
int StackSize = 0;

double doubleValue{1.1}; // first value to push
int intValue{1}; // first value to push
char charValue{'a'};
```

Setting up a variable for the size of our Stacks and initializing the first value going into our Stacks.

```
StackSize = 5;
cout << "\nPushing " << StackSize << " elements onto doubleStack\n";

// push 5 doubles onto doubleStack
for (int i = 0; i < StackSize; ++i)
{
      doubleStack.push(doubleValue);
      cout << doubleValue << "\t";
      doubleValue += 1.1;
}

cout << "\n\nPopping elements from doubleStack\n";

// pop elements from doubleStack
while (!doubleStack.isEmpty())
{
      cout << doubleStack.top() << "\t"; // display top element
      doubleStack.pop(); // remove top element
}
```

`push()` is a member function of our object `doubleStack`.

`isEmpty()` is a member function of our object `doubleStack`.

`top()` and `pop()` are member functions of our object `doubleStack`

```
StackSize = 10;
cout << "\n\n\nPushing " << StackSize << " elements onto intStack\n";

// push 10 integers onto intStack
for (int i = 0; i < StackSize; ++i)
{
	intStack.push(intValue);
	cout << intValue++ << "\t";
}

cout << "\n\nPopping elements from intStack\n";

// pop elements from intStack
while (!intStack.isEmpty())
{
	cout << intStack.top() << "\t";
	intStack.pop();
}
```

Because `intStack` was instantiated using class `Stack`, it also has member functions `push()`, `isEmpty()`, `top()` and `pop()`.

```
StackSize = 3;
cout << "\n\n\nPushing " << StackSize << " elements onto charStack\n";

// push 3 integers onto charStack
for (int i = 0; i < StackSize; ++i)
{
      charStack.push(charValue);
      cout << charValue++ << "\t";
}

cout << "\n\nPopping elements from charStack\n";

// pop elements from charStack
while (!charStack.isEmpty())
{
      cout << charStack.top() << "\t";
      charStack.pop();
}
```

Because `charStack` was instantiated using class `Stack`, it also has member functions `push()`, `isEmpty()`, `top()` and `pop()`.

Class Templates

`charStack`, `intStack` and `doubleStack` all have access to the same functions because they were instantiated from the same class `Stack`.

So how did class `Stack` create a vector of `char`acters, `int`s and `double`s?

Class `Stack` is a template class.

First, we set up the include guard in our `Stack.h` file.

We need the include for vector since we are using a vector as our "stack".

We want this to be a template class so we add

`template <typename T>`

right before class `Stack` to make our class a template.

```
// Stack class template.
#ifndef STACK_H
#define STACK_H
#include <vector>

template<typename T>
class Stack
{

// class body on next slide

};

#endif
```

```
template<typename T>
class Stack
{
    public:
        const T &top()
        {
            return stack.front();
        }

        void push(const T& pushValue)
        {
            stack.insert(stack.begin(), pushValue);
        }

        void pop()
        {
            stack.erase(stack.begin());
        }

        bool isEmpty() const
        {
            return stack.empty();
        }

        int size() const
        {
            return stack.size();
        }

    private:
        std::vector<T> stack;
};

#endif
```

Private data member `stack` is a `vector` of type T where T is our template substitution. So when we instantiated our objects, we passed in the type we wanted substituted for T.

```
Stack<double> doubleStack;
Stack<int> intStack;
Stack<char> charStack;
```

The public member functions then take advantage of vector's abilities to emulate stack behavior.

Notice that function `top()` returns the first value of the vector using `front()`. The T in

```
const T &top()
```

will be replaced by `double` when `doubleStack` is instantiated and by `int` when `intStack` is instantiated and by `char` when `charStack` is instantiated.

```
template<typename T>
class Stack
{
    public:
        const T &top()
        {
            return stack.front();
        }

        void push(const T &pushValue)
        {
            stack.insert(stack.begin(), pushValue);
        }

        void pop()
        {
            stack.erase(stack.begin());
        }

        bool isEmpty() const
        {
            return stack.empty();
        }

        int size() const
        {
            return stack.size();
        }

    private:
        std::vector<T> stack;
};

#endif
```

Notice that function `push()` has a parameter `pushValue` of type T. The T in

`void push(const T &pushValue)`

will be replaced by `double` when `doubleStack` is instantiated and by `int` when `intStack` is instantiated and by `char` when `charStack` is instantiated.

```
template<typename T>
class Stack
{
    public:
        const T &top()
        {
            return stack.front();
        }

        void push(const T &pushValue)
        {
            stack.insert(stack.begin(), pushValue);
        }

        void pop()
        {
            stack.erase(stack.begin());
        }

        bool isEmpty() const
        {
            return stack.empty();
        }

        int size() const
        {
            return stack.size();
        }

    private:
        std::vector<T> stack;
};

#endif
```

Be careful when using templates – don't try to use a data type when instantiating objects using a class template that the functions in the template cannot handle.

In this example, the data type that T will be replaced with must be a type that you can create a vector of and that those vector functions can handle.