Object-Oriented Programming (OOP)
Lecture No. 44

Example

```
class DivideByZero {
public:
        DivideByZero() {
        }
};
int Quotient(int a, int b){
        if(b == 0){
                throw DivideByZero();
        }
        return a / b;
}
```

main Function

```
int main() {
        try{    …
                quot = Quotient(a,b);
                …
        }
        catch(DivideByZero) {
                …
        }
        return 0;
}
```

Stack Unwinding

- The flow control of throw is referred to as stack unwinding
- Stack unwinding is more complex than return statement
- Return can be used to transfer the control to the calling function only
- Stack unwinding can transfer the control to any function in nested function calls

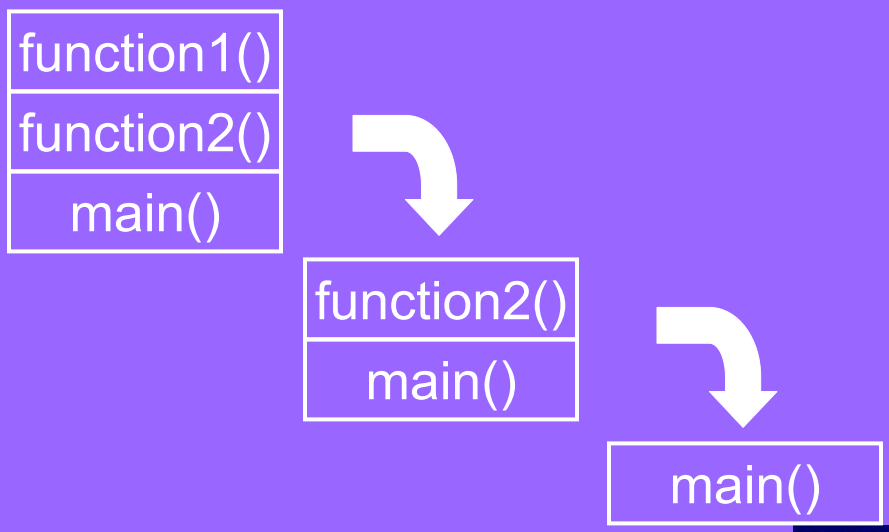
Stack Unwinding

- All the local objects of a executing block are destroyed when an exception is thrown
- Dynamically allocated memory is not destroyed automatically
- If no catch handler catches the exception the function terminate is called, which by default calls function abort

Example

```
void function1() {          ...
        throw Exception(); …
}
void function2() {…
        function1();…
}
int main() {
        try{
                function2();
        } catch( Exception ) { }
        return 0;
}
```

Function-Call Stack



Stack Unwinding

- The stack unwinding is also performed if we have nested try blocks

Example

```
int main( ) {
        try {
                try {
                        throw 1;
                }
                catch( float ) { }
        }
        catch( int ) { }
        return 0;
}
```

Stack Unwinding

- Firstly the catch handler with float parameter is tried
- This catch handler will not be executed as its parameter is of different type – no coercion
- Secondly the catch handler with int parameter is tried and executed

Catch Handler

- We can modify this to use the object to carry information about the cause of error
- The object thrown is copied to the object given in the handler
- Use the reference in the catch handler to avoid problem caused by shallow copy

Example

```
class DivideByZero {
        int numerator;
public:
        DivideByZero(int i) {
                numerator = i;
        }
        void Print() const{
                cout << endl << numerator
                << " was divided by zero";
        }
};
```

Example

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

Body of main Function

```
for ( int i = 0; i < 10; i++ ) {
        try  {
                GetNumbers(a, b);
                quot = Quotient(a, b);        ...
        } catch(DivideByZero & obj) {
                obj.Print();
                i--;
        }
}
cout << "\nSum of ten quotients is "
        << sum;
```

Output

```
Enter two integers
10
10
Quotient of 10 and 10 is 1
Enter two integers
10
0
10 was divided by zero
...
// there will be sum of exactly ten quotients
```

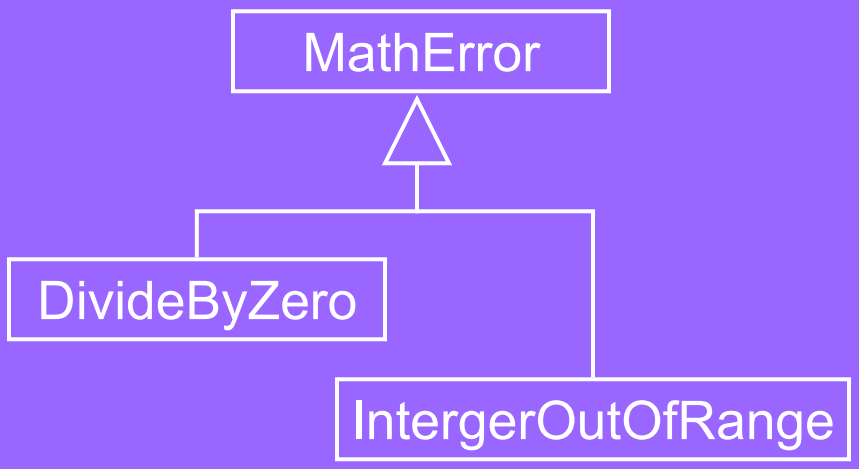
Catch Handler

- The object thrown as exception is destroyed when the execution of the catch handler completes

Avoiding too many Catch Handlers

- There are two ways to catch more then one object in a single catch handler
  - Use inheritance
  - Catch every exception

Inheritance of Exceptions



Grouping Exceptions

```
try{
        …
}
catch(DivideByZero){
        …
}
catch(IntergerOutOfRange){
        …
}
catch (InputStreamError){
}
```

Example–With Inheritance

```
try{
        …
}
catch (MathError){
}
catch (InputStreamError){
}
```

Catch Every Exception

- C++ provides a special syntax that allows to catch every object thrown

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

Re-Throw

- A function can catch an exception and perform partial handling
- Re-throw is a mechanism of throw the exception again after partial handling

```
throw; /*without any expression*/
```

Example

```
int main ( ) {
        try {
                Function();
        }
        catch(Exception&) {
                ...
        }
        return 0;
}
```

Example

```
void Function( ) {
        try {
                /*Code that might throw
                        an Exception*/
        } catch(Exception&){
                if( can_handle_completely ) {
                        // handle exception
                }  else  {
                        // partially handle exception
                        throw; //re-throw exception
}       }       }
```

Order of Handlers

- Order of the more then one catch handlers can cause logical errors when using inheritance or catch all

Example

```
try{
        …
}
catch (…) {                    …
}
catch ( MathError )  {         …
}
catch ( DivideByZero ) {       …
}
// last two handler can never be invoked
```