Exception Handling

Techniques for Error Handling

- Abnormal termination
- Graceful termination
- Return the illegal value
- Return error code from a function
- Exception handling

Example – Abnormal Termination

```
void GetNumbers( int &a, int &b ) {
        cout << “\nEnter two integers”;
        cin >> a >> b;
}
int Quotient( int a, int b ){
        return a / b;
}
void OutputQuotient( int a, int b, int quo ) {
        cout << “Quotient of ” << a << “ and ”
                << b << “ is ” << quo << endl;
}
```

Example – Abnormal Termination

```
int main(){
        int sum = 0, quot;
        int a, b;
        for (int i = 0; i < 10; i++){
                GetNumbers(a,b);
                quot = Quotient(a,b);
                sum += quot;
                OutputQuotient(a,b,quot);
        }
        cout << "\nSum of ten quotients is "<< sum;
        return 0;
}
```

Output

```
Enter two integers
10
10
Quotient of 10 and 10 is 1
Enter two integers
10
0
Program terminated abnormally
```

Graceful Termination

- Program can be designed in such a way that instead of abnormal termination, that causes the wastage of resources, program performs clean up tasks

Example – Graceful Termination

```
int Quotient (int a, int b ) {
        if(b == 0){
                cout << “Denominator can’t “
                << “ be zero” << endl;
                // Do local clean up
                exit(1);
        }
        return a / b;
}
```

Output

Enter two integers

10

10

Quotient of 10 and 10 is 1

Enter two integers

10

0

Denominator can’t be zero

Error Handling

- The clean-up tasks are of local nature only
- There remains the possibility of information loss

Example – Return Illegal Value

```
int Quotient(int a, int b){
        if(b == 0)
                b = 1;
        OutputQuotient(a, b, a/b);
        return a / b ;
}
int main() {
        int a,b,quot;           GetNumbers(a,b);
        quot = Quotient(a,b);
        return 0;
}
```

Output

```
Enter two integers
10
0
Quotient of 10 and 1 is 10
```

Error Handling

- Programmer has avoided the system crash but the program is now in an inconsistent state

Example – Return Error Code

```
bool Quotient ( int a, int b, int & retVal ) {
        if(b == 0){
                return false;
        }
        retVal = a / b;
        return true;
}
```

Part of main Function

```
for(int i = 0; i < 10; i++){
        GetNumbers(a,b);
        while ( ! Quotient(a, b, quot) ) {
                cout << “Denominator can’t be ”
        << “Zero. Give input again \n”;
                GetNumbers(a,b);
        }
        sum += quot;
        OutputQuotient(a, b, quot);
}
```

Output

Enter two integers
10
0
Denominator can’t be zero. Give input again.
Enter two integers
10
10
Quotient of 10 and 10 is 1
…//there will be exactly ten quotients

Error Handling

- Programmer sometimes has to change the design to incorporate error handling
- Programmer has to check the return type of the function to know whether an error has occurred

Error Handling

- Programmer of calling function can ignore the return value
- The result of the function might contain illegal value, this may cause a system crash later

Program's Complexity Increases

- The error handling code increases the complexity of the code
  - Error handling code is mixed with program logic
  - The code becomes less readable
  - Difficult to modify

Example

```
int main() {
        function1();
        function2();
        function3();

        return 0;
}
```

Example

```
int main(){
        if( function1() ) {
                if( function2() ) {
                        if( function3() ) {
                                ...
                        }
                        else    cout << “Error Z has occurred”;
                }
                else    cout << “Error Y has occurred”;
        }
        else    cout << “Error X has occurred”;
        return 0;
}
```

Exception Handling

- Exception handling is a much elegant solution as compared to other error handling mechanisms
- It enables separation of main logic and error handling code

Exception Handling Process

- Programmer writes the code that is suspected to cause an exception in try block
- Code section that encounters an error throws an object that is used to represent exception
- Catch blocks follow try block to catch the object thrown

Syntax - Throw

- The keyword throw is used to throw an exception
- Any expression can be used to represent the exception that has occurred

```
throw X;
throw (X);
```

Examples

```
int a;
Exception obj;
throw 1;                    // literal
throw (a);                  // variable
throw obj;                  // object
throw Exception();
                // anonymous  object
throw 1+2*9;
                // mathematical expression
```

Throw

- Primitive data types may be avoided as throw expression, as they can cause ambiguity
- Define new classes to represent the exceptions that has occurred
  - This way there are less chances of ambiguity

Syntax – Try and Catch

```
int main () {
        try {
                ...
        }
        catch ( Exception1 ) {
                ...
        }
        catch ( Exception2 obj ) {
                ...
        }
        return 0;
}
```

Catch Blocks

- Catch handler must be preceded by a try block or an other catch handler
- Catch handlers are only executed when an exception has occurred
- Catch handlers are differentiated on the basis of argument type

Catch Handler

- The catch blocks are tried in order they are written
- They can be seen as switch statement that do not need break keyword

Example

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

Body of main Function

```
for(int i = 0; i < 10; i++) {
        try{
                GetNumbers(a,b);
                quot = Quotient(a,b);
                OutputQuotient(a,b,quot); sum += quot;
        }
        catch(DivideByZero) {
                i--;
                cout << "\nAttempt to divide
                        numerator with zero";
        }
}
```

Output

```
Enter two integers
10
10
Quotient of 10 and 10 is 1
Enter two integers
10
0
Attempt to divide numerator with zero
...
// there will be sum of exactly ten quotients
```

Catch Handler

- The catch handler catches the DivideByZero object through anonymous object
- Program logic and error handling code are separated
- We can modify this to use the object to carry information about the cause of error

Separation of Program Logic and Error Handling

```
int main() {
        try {
                function1();
                function2();
                function3();
        }
        catch( ErrorX) { ... }
        catch( ErrorY) { ... }
        catch( ErrorZ) { ... }
        return 0;
}
```

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

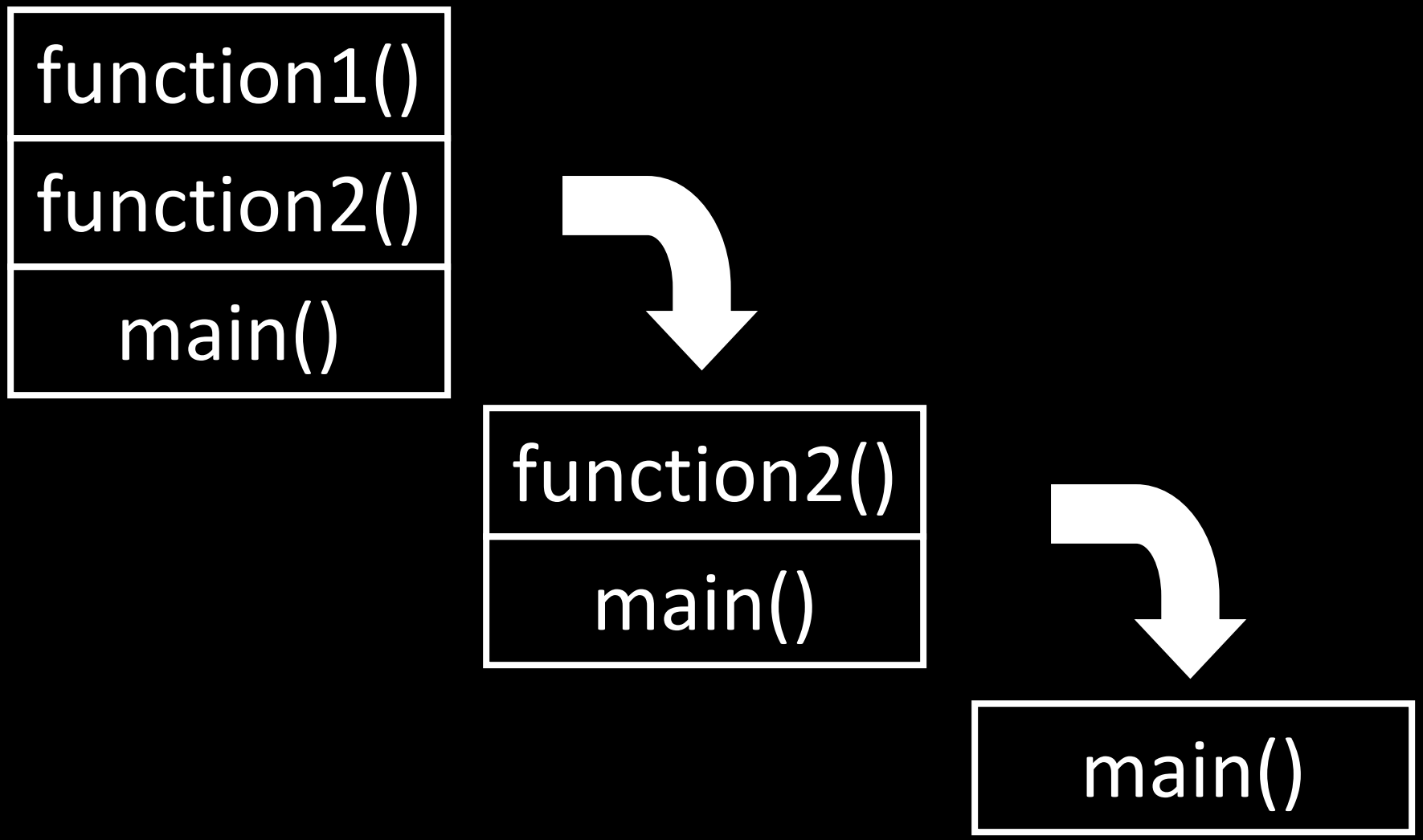
Stack Unwinding

- 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

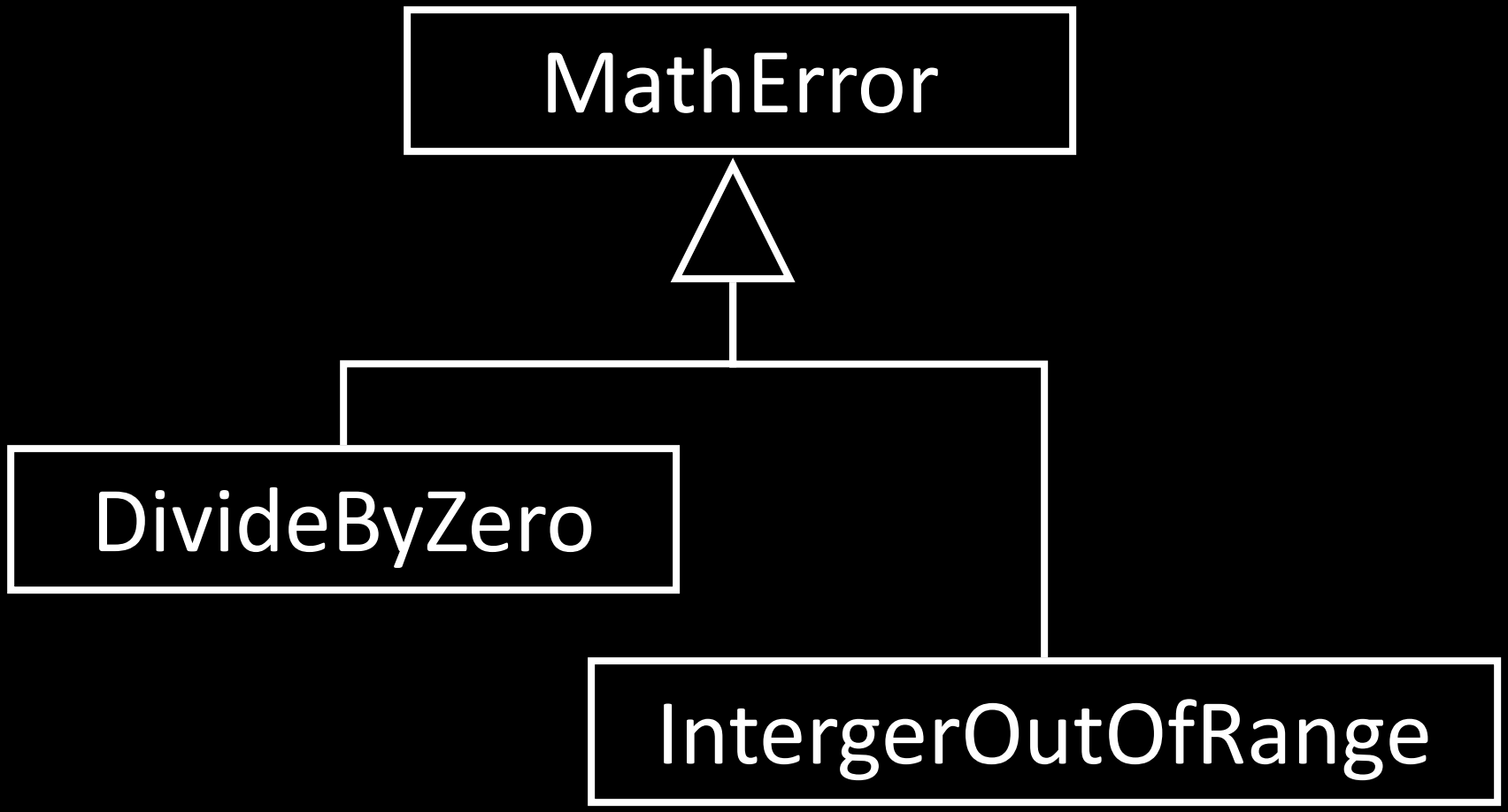
Catch Handler

- 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
```

Resource Management

- Function acquiring a resource must properly release it
- Throwing an exception can cause resource wastage

Example

```
int function1(){
        FILE *fileptr =
                fopen(“filename.txt”,“w”);
        …
        throw exception();
        …
        fclose(fileptr);
        return 0;
}
```

Resource Management

- In case of exception the call to close will be ignored

First Attempt

```
int function1(){
        try{
                FILE *fileptr = fopen(“filename.txt”,“w”);
                fwrite(“Hello  World”,1,11,fileptr);
                ...
                throw exception();
                fclose(fileptr);
        } catch(...) {
                fclose(fileptr);
                throw;
        }
        return 0;
}
```

Resource Management

- There is code duplication

Second Attempt

```
class FilePtr{
        FILE * f;
public:
        FilePtr(const char *name,
                        const char * mode)
                { f = fopen(name, mode);}
        ~FilePtr()          {  fclose(f);       }
   };
```

Example

```
int function1(){
        FilePtr file(“filename.txt”,“w”);
        fwrite(“Hello  World”,1,11,file);
        throw exception();
        ...
        return 0;
}
```

Resource Management

- The destructor of the FilePtr class will close the file
- Programmer does not have to close the file explicitly in case of error as well as in normal case

Exception in Constructors

- Exception thrown in constructor cause the destructor to be called for any object built as part of object being constructed before exception is thrown
- Destructor for partially constructed object is not called

Example

```
class Student{
        String FirstName;
        String SecondName;
        String EmailAddress;
        …
}
```

- If the constructor of the SecondName throws an exception then the destructor for the First Name will be called

Exception in Initialization List

- Exception due to constructor of any contained object or the constructor of a parent class can be caught in the member initialization list

Example

```
Student::Student (String aName) :
    name(aName)
/*The constructor of String can throw a
    exception*/
{
        …
}
```

Exception in Initialization List

- The programmer may want to catch the exception and perform some action to rectify the problem

Example

```
Student::Student (String aName)
    try
        : name(aName) {
        ...
    }
    catch(…) {
    }
```

Exceptions in Destructors

- Exception should not leave the destructor
- If a destructor is called due to stack unwinding, and an exception leaves the destructor then the function std::terminate() is called, which by default calls the std::abort()

End of the Course!