Advanced Object Oriented Programming

Exception Handling

Seokhee Jeon
Department of Computer Engineering
Kyung Hee University
jeon@khu.ac.kr

Errors

- User error
  - E.g., bad input data
- Hardware error
  - E.g., a disk read error
- Programming error
  - E.g, a bad pointer

➔ To become robust programs, programs need to handle potential errors and abnormal conditions

Traditional Error Handling

- Uses the return value from a function to communicate status
- If the function is successful, it returns 0
- If an error occurs, it returns a unique number to identify the error
- This is true even for the *main()* function
  - Who checks the return value of main()?

Traditional exception handling approach

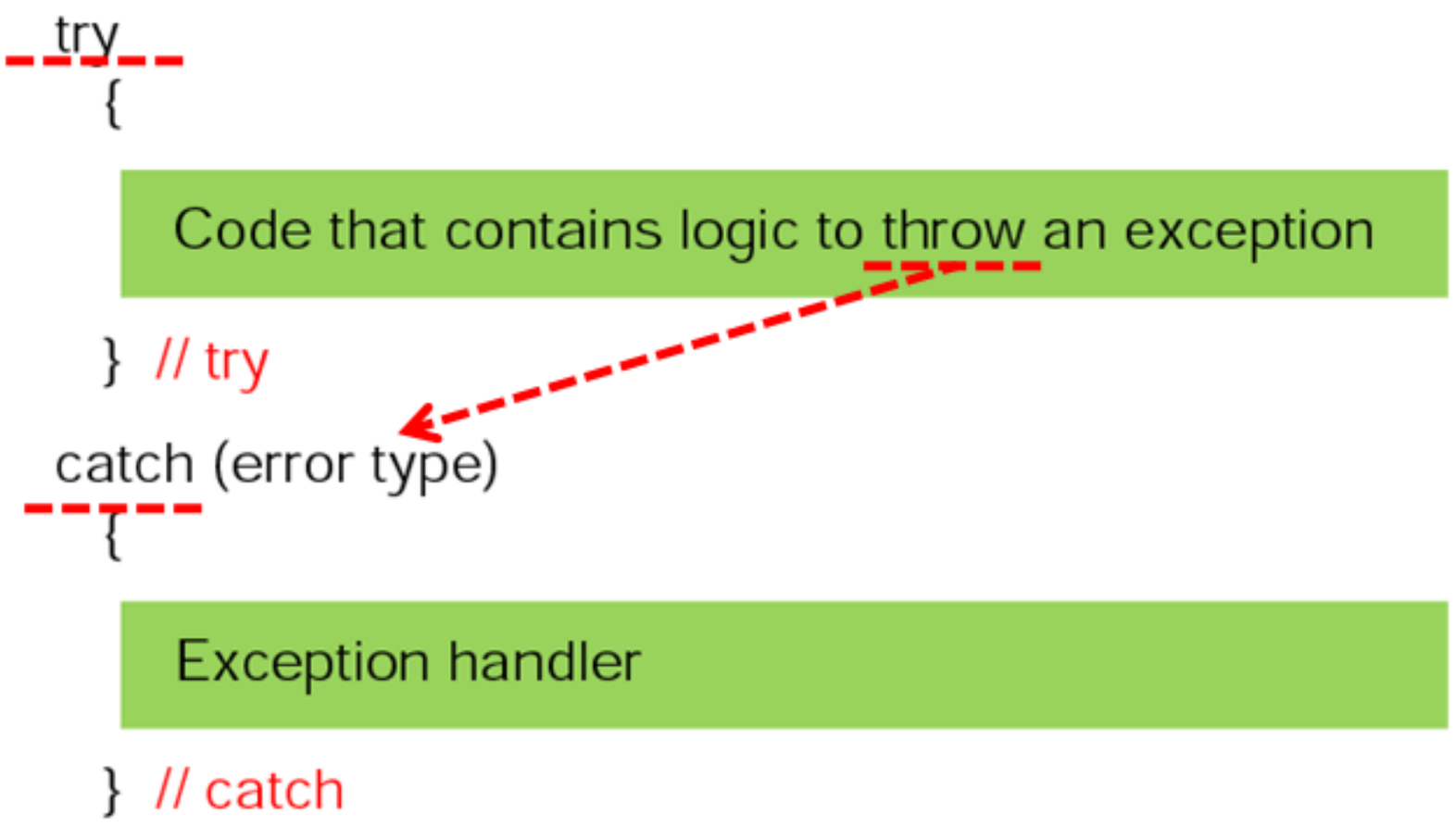
```
#include <iostream>
using namespace std;

int main ()
{
        cout << "Enter the dividend: ";
        double dividend;
        cin  >> dividend;
        cout << "Enter the divisor:  ";
        double divisor;
        cin  >> divisor;

        if (divisor == 0)
          {
           cout << "**Error 100: divisor 0\n";
           return 1;          // Tell OS program failed
          } // if

        double quotient = dividend / divisor;
        cout << "Quotient is:    " << quotient << endl;
        return 0;
}       // main
```

```
/*                Results:
Run 1:
                  Enter the dividend: 7
                  Enter the divisor:  0
                  **Error 100: divisor 0
Run 2:
                  Enter the dividend: 7
                  Enter the divisor:  3
                  Quotient is:    2.33333
*/
```

Using Exception Handling

- Systematic approach for error handling
- An ***exception*** is an event that signals the occurrence of an error
- This separates the code that may generate the event (error) from the code that handles it
  - The error detection logic is coded in a special construct called the ***try* statement (block)**
  - The error handling logic is coded in another construct called the ***catch* statement (block)**

C++ Exception handling : try & catch



Using Exception Handling

- The *try* statement block ***throws*** an exception that is caught by the *catch* statement
  - The statements after the throw statement inside the try block are not executed any more
  - If no error is detected, the *try* statement executes normally and the *catch* statement is ignored
- An exception is thrown by using a **typed object (a standard type or a class instance)**, which is used to transfer information about the errors occurred

C++ Exception handling : try & catch

```
#include <iostream>
using namespace std;
int main ()
{
        cout << "Enter the dividend: ";
        double dividend;
        cin  >> dividend;
        cout << "Enter the divisor:  ";
        double divisor;
        cin  >> divisor;

        try
          {
          if (divisor == 0.00)
              throw divisor;
          double quotient = dividend / divisor;
          cout << "Quotient is:     " << quotient << endl;
          } // try

        catch (double& error)
          {
          cout << "**Error 100: divisor 0\n";
          return 1;
          } // catch
        return 0;
} // main
```

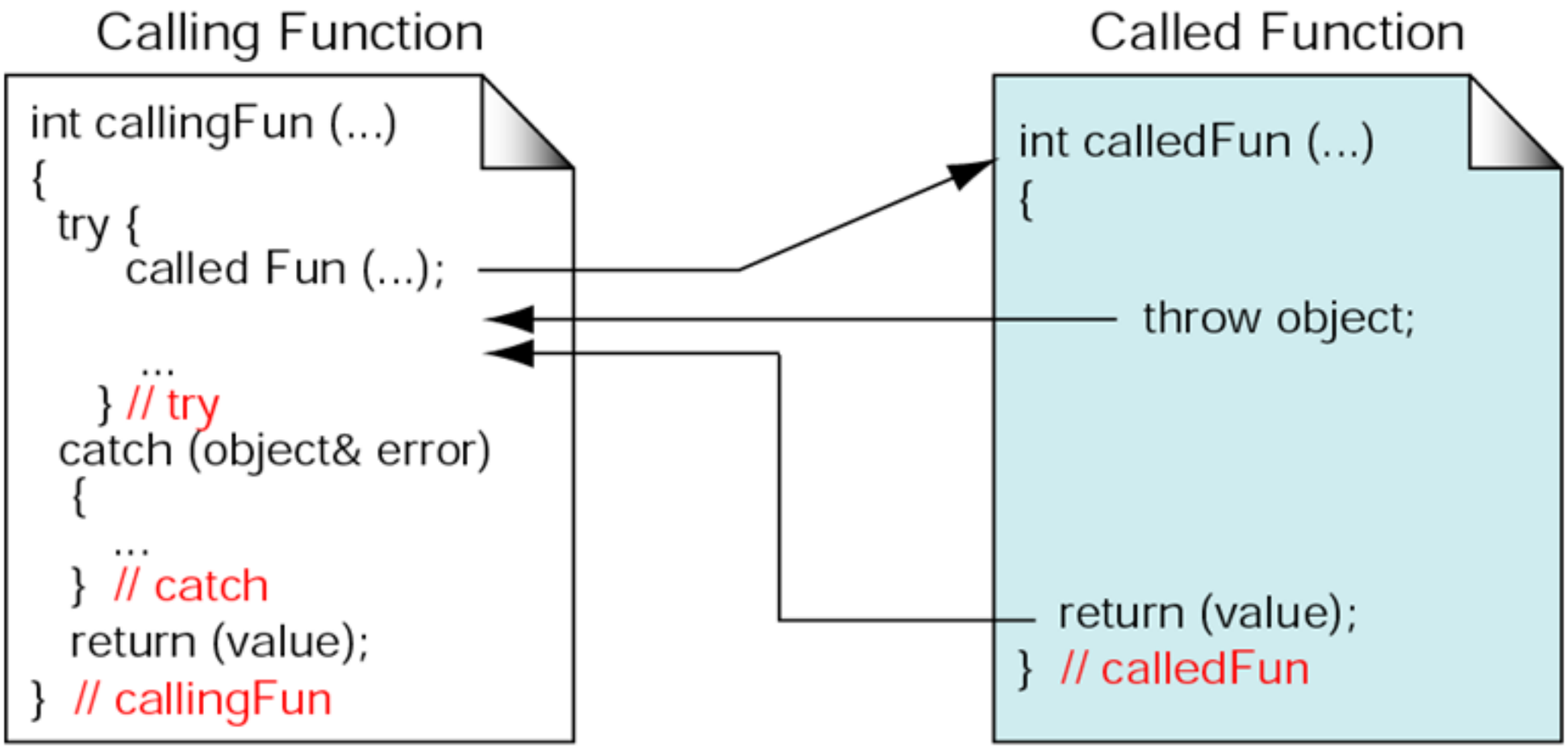
```
/*              Results:
Run 1:
                Enter the dividend: 7
                Enter the divisor:  0
                **Error 100: divisor 0
Run 2:
                Enter the dividend: 7
                Enter the divisor:  3
                Quotient is:    2.33333
*/
```

C++ Exception handling : try & catch

```
#include <iostream>
using namespace std;
int main ()
{
        cout << "Enter the dividend: ";
        double dividend;
        cin  >> dividend;
        cout << "Enter the divisor:  ";
        double divisor;
        cin  >> divisor;

        try
          {
          if (divisor == 0.00)
              throw divisor;
          double quotient = dividend / divisor;
          cout << "Quotient is:     " << quotient << endl;
         } // try

        catch (double& error)
          {
          cout << "**Error 100: divisor 0\n";
          return 1;
         } // catch
        return 0;
} // main
```

```
/*              Results:
Run 1:
                Enter the dividend: 7
                Enter the divisor:  0
                **Error 100: divisor 0
Run 2:
                Enter the dividend: 7
                Enter the divisor:  3
                Quotient is:    2.33333
*/
```

Throwing an exception in a separate function



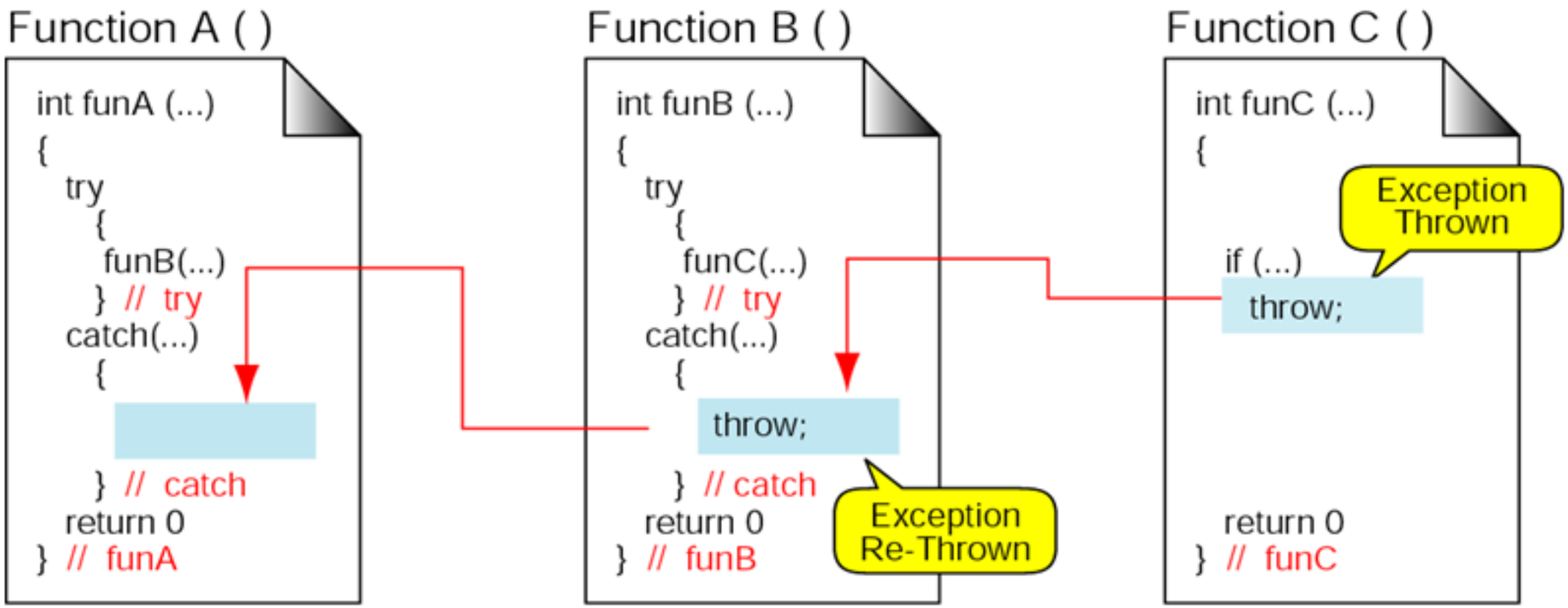
C++ Exception handling : try & catch

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

double divide (double quot, double dvsr);

int main ()
{
        cout << "Enter the dividend: ";
        double dividend;
        cin  >> dividend;
        cout << "Enter the divisor:  ";
        double divisor;
        cin  >> divisor;

        try
          {
          double quotient = divide (dividend, divisor);
          cout << "Quotient is:     " << quotient << endl;
          } // try

        catch (double& error)
          {
          cout << "**Error 100: divisor 0\n";
          return 1;
          } // catch
        cout << "End of Exception Handling Test\n";
        return 0;
}       // main

double divide (double dvnd, double dvsr)
{
        if (dvsr == 0)
        throw dvsr;
        return dvnd / dvsr;
}       // divide
```

```
/*              Results:
Run 1:
                Enter the dividend: 7
                Enter the divisor:  0
                **Error 100: divisor 0
Run 2:
                Enter the dividend: 7
                Enter the divisor:  3
                Quotient is:    2.33333
                End of Exception Handling Test
*/
```

Re-throwing an exception

- An exception handler can **re-throw** the exception to the higher level function that called it
- A re-throw statement uses *throw* without an object identifier



Multiple Errors

- A function can test for different types of errors and throw different results depending on the error
- The calling function includes a series of *catch* statements, one for each error
- C++ chooses as the exception handler the *first catch statement* whose parameter matches the type of object thrown

Multiple error handling

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

double divide (double dividend, double divisor);

int main ()
{
        cout << "Enter the dividend: ";
        double dividend;
        cin  >> dividend;
        cout << "Enter the divisor:  ";
        double divisor;
        cin  >> divisor;

        try
          {
           double quotient = divide (dividend, divisor);
           cout << "Quotient is : " << quotient << endl;
          } // try

        catch (float& zeroError)
          {
           cout << "**Error 100: divisor 0\n";
           return 100;
          } // DivByZero

        catch (double& negError)
          {
           cout << "**Error 101: negative divisor\n";
           return 101;
          } // DivByNeg

        cout << "End of Exception Handling Test\n";
        return 0;
}       // main
```

```
double divide (double dvnd, double dvsr)
{
        float  zeroError = 0;
        double negError  = -1;

        if (dvsr == 0)
           throw zeroError;
        if (dvsr < 0)
           throw negError;
        return dvnd / dvsr;
}       // divide
```

```
/*              Results
Run 1:
                Enter the dividend: 10
                Enter the divisor:  0
                **Error 100: divisor 0
Run 2:
                Enter the dividend: 10
                Enter the divisor:  -1
                **Error 101: negative divisor
Run 3:
                Enter the dividend: 10
                Enter the divisor:  3
                Quotient is : 3.33333
                End of Exception Handling Test
*/
```

Generic Handler

- Sometimes we cannot determine all of their errors that may occur or do not need to provide unique handlers for each error
- If an error occurs for which their is no handler, the system terminates the program
- To prevent system aborts, we can include a generic handler in the program
- This can be done by a *generic catch statement*, a *catch* statement that uses ellipses (...)

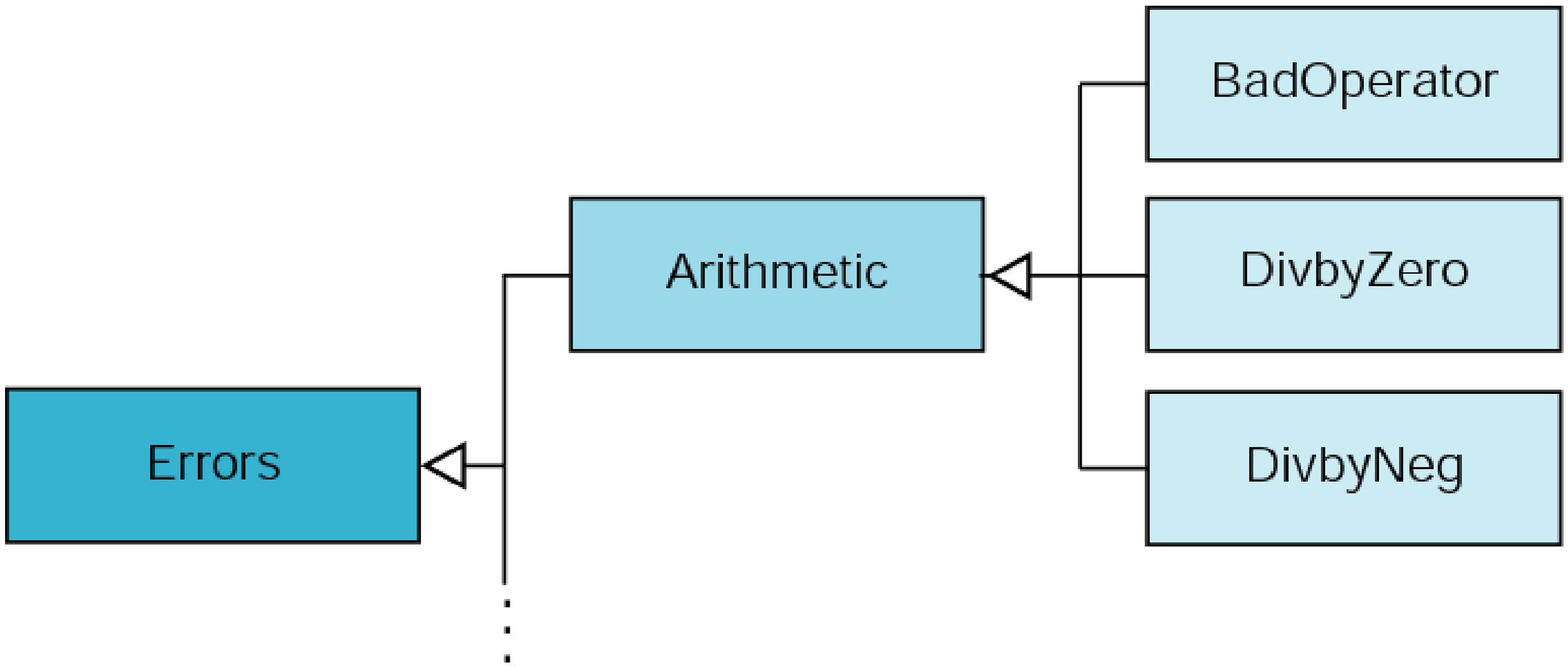
Generic error handling

```
try
  {
   ...
  }
catch (Object1& e1)
  {
   ...
  }
catch (Object2& e2)
  {
   ...
  }
catch (...)
  {
   ...          //  Code for handling generic error
  }
```

Error Classes

- The standard type error handlers work well in simple situations
- For more complex error handing, especially for generalized software functions, *error class* provide a more powerful and elegant solutions
- Error classes allow us to categorize errors and create a hierarchy of classes
  - Each class represents one type of error

Error Class Design



Error class design

```
#include <stdexcept>
using namespace std;
class Error
{
        public:
        virtual void printMessage ()
          {cout << "**Error: type Error\n";}
};      // Error
class Arithmetic: public Error
{
        public:
        virtual void printMessage ()
          {cout << "**Error: type Arithmetic\n";}
};      // Arithmetic
class DivbyZero: public Arithmetic
{
        public:
        virtual void printMessage ()
          {cout << "**Error: 100 divisor 0\n";}
};      // DivbyZero
class DivbyNeg: public Arithmetic
{
        public:
        virtual void printMessage ()
          {cout << "**Error: 101 negative divisor\n";}
};      // DivbyZero
class BadOperator: public Arithmetic
{
        public:
        virtual void printMessage ()
          {cout << "**Error: 102 invalid operator\n";}
};// BadOperator
```

```
double math (char oper, double data1, double data2)
{
                double result;
                  switch (oper)

                    {
                     case '+': result = data1 + data2;
                               break;
                     case '-': result = data1 - data2;
                               break;
                     case '*': result = data1 * data2;
                               break;
                     case '/': if (data2 == 0 )
                                 throw DivbyZero ();
                               if (data2 < 0 )
                                 throw DivbyNeg ();
                               result = data1 / data2 ;
                               break;
                     default:  throw BadOperator () ;
                               break;
                    } // switch
              return result;
}             // math
```

Error class implementation

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

#include "p15-05.h"            // Error class hierarchy
#include "p15-06.h"            // Math function
#define FLUSH while (cin.get() != '\n')

int main ()
{
        cout << "Begin Error class demonstration\n";

        cout << "Enter the first data:  " ;
        double data1;
        cin >> data1;
        cout << "Enter the second data: " ;
        double data2;
        cin >> data2;
        cout << "Enter the operator:    " ;
        char oper;
        cin  >> oper;
        FLUSH;

        double result;
        try
          {
          result = math (oper, data1 , data2 );
          cout << "result: "<< result << endl;
          } // try

        catch (Error& error)
          {
          error.printMessage() ;
          return 100;
          } // catch
        cout << "Normal end of demonstration\n";
        return 0;
}       // main
```

Advantages of Using Error Classes

- We include an error message for each type in the appropriate class and throw different object for different errors
- The error messages are printed in virtual functions

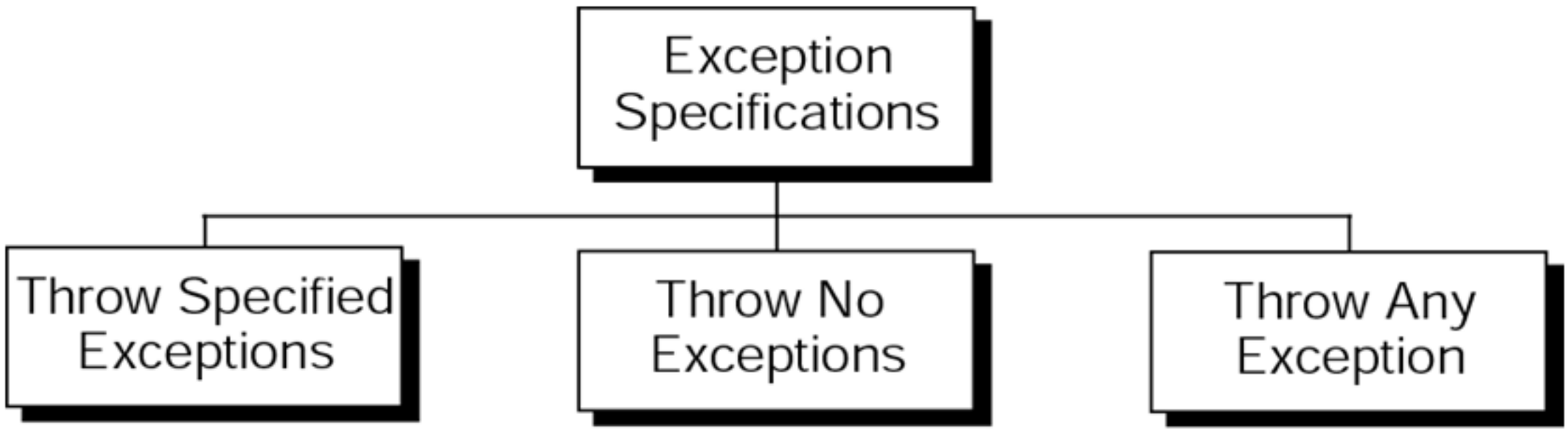
➔ Using polymorphism, we can create a "**generic**" handler

- use only one *catch* statement to handle multiple errors

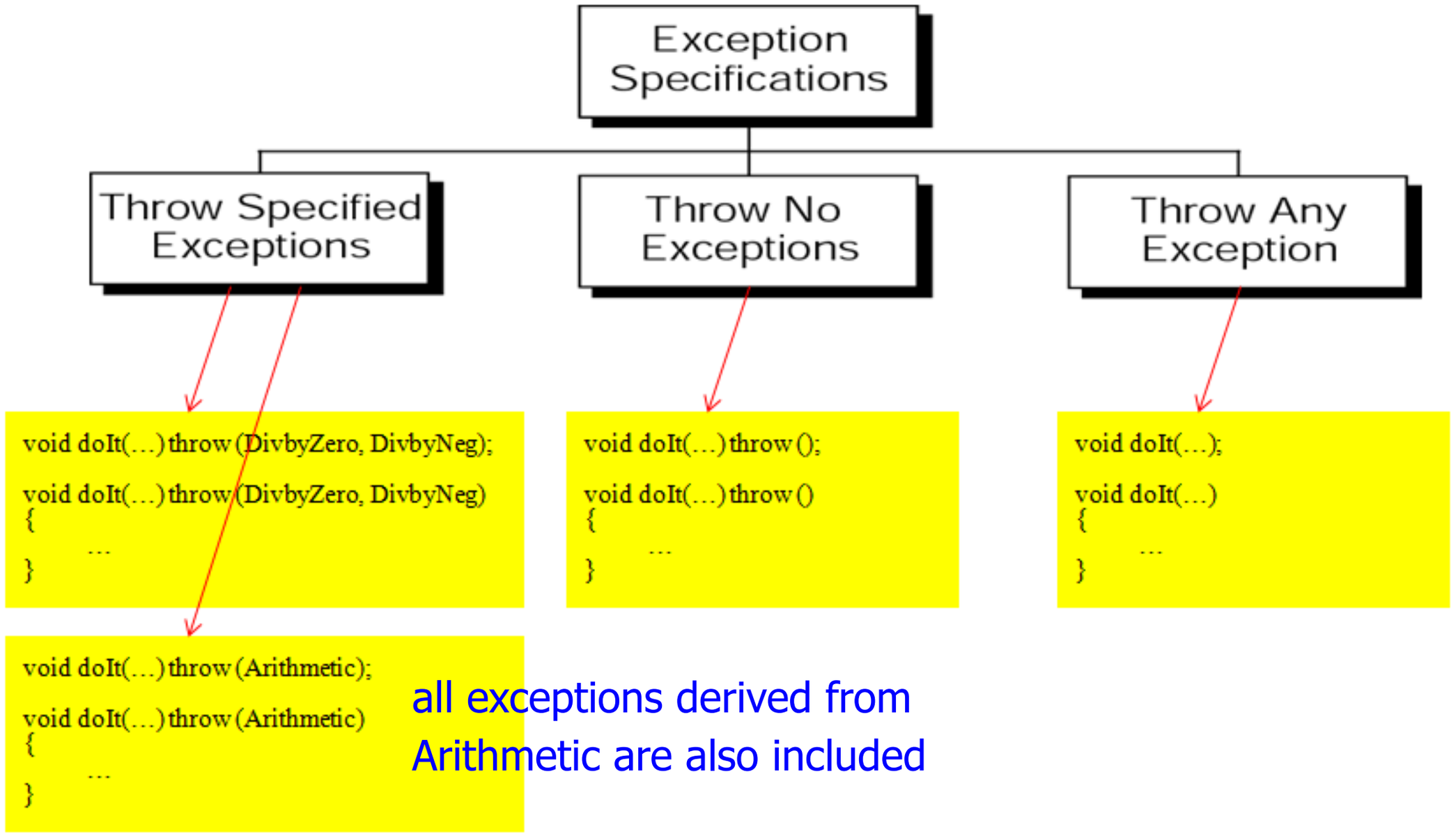
Exception Specification

- By default, all functions are allowed to throw any type of exception
- To control what exceptions a function is allowed to throw, we can add an *exception specification* to the declaration and definition of the function
- Any attempt to throw an unauthorized exception results in a call to a special system function, *unexpected*, that aborts the program

Exception specifications



Exception specifications



The *unexpected* Exception

- If a function encounters an exception that is not part of its exception specification, then the exception hander calls the *unexpected()* library function
- The unexpected function calls, by default, another function called *terminate()*, which by default calls the *abort()* function to abort the program
- The default action is undesirable since it terminates the program ungracefully

Change Action for *unexpected*

- We can change the default action of the unexpected function by calling the ***set_unexpected*** function
- The parameter of *set_unexpected* is the name of a handler function that we write
- The name of a function is a pointer to that function (See Appendix M)

```
void myTerminateHandler()
{
...
} // myTerminateHandler;
oldHandler = set_unexpected (myTerminateHandler);
```

Change Action for *unexpected*

- We can change the default action of the unexpected function by calling the ***set_unexpected*** function
- The parameter of *set_unexpected* is the name of a handler function that we write
- The name of a function is a pointer to that function (See Appendix M)

```
void myTerminateHandler()
{
...
} // myTerminateHandler;
oldHandler = set_unexpected (myTerminateHandler);
```

```
// prototype of set_unexpected
typedef void (*unexpected_function) ( );

unexpected_function set_unexpected (
      unexpected_function unexp_func
);
```

Changing Default for *terminate*

- We can change the default action of the *terminate* function by calling the ***set_terminate*** function
- The parameter of *set_terminate* is the name of a handler function that we write

abort versus *exit*

- A program can be terminated abnormally by using one of the two system functions: *abort* or *exit*
- The *abort* function is very undesirable because it does not terminate gracefully
  - No files are closed and only cryptic system messages, if any, are displayed
- On the other hand, the *exit* function terminates gracefully

Exceptions in Classes

Exceptions in Constructors

- The constructor must handle critical exceptions, such as memory allocation
  - The destructor is never called if a constructor is terminated abnormally at the middle of its execution, so the mess is not cleaned out
- C++ even allows us to call the *try* function with the initialization list

Exceptions in Destructors

- If, during the execution of the destructor, it fails and throws an exception, the system immediately terminates the program
- To prevent the destructor from throwing any exception, add an exception specification ***throw ()***

Exceptions in Class (Constructor, Destructor)

```
func::func() : foo()
{
    try {...}
    catch (...) // will NOT catch exceptions thrown from foo constructor
    { ... }
}
```

vs.

```
func::func()
    try : foo() {...}
    catch (...) // will catch exceptions thrown from foo constructor
    { ... }
```

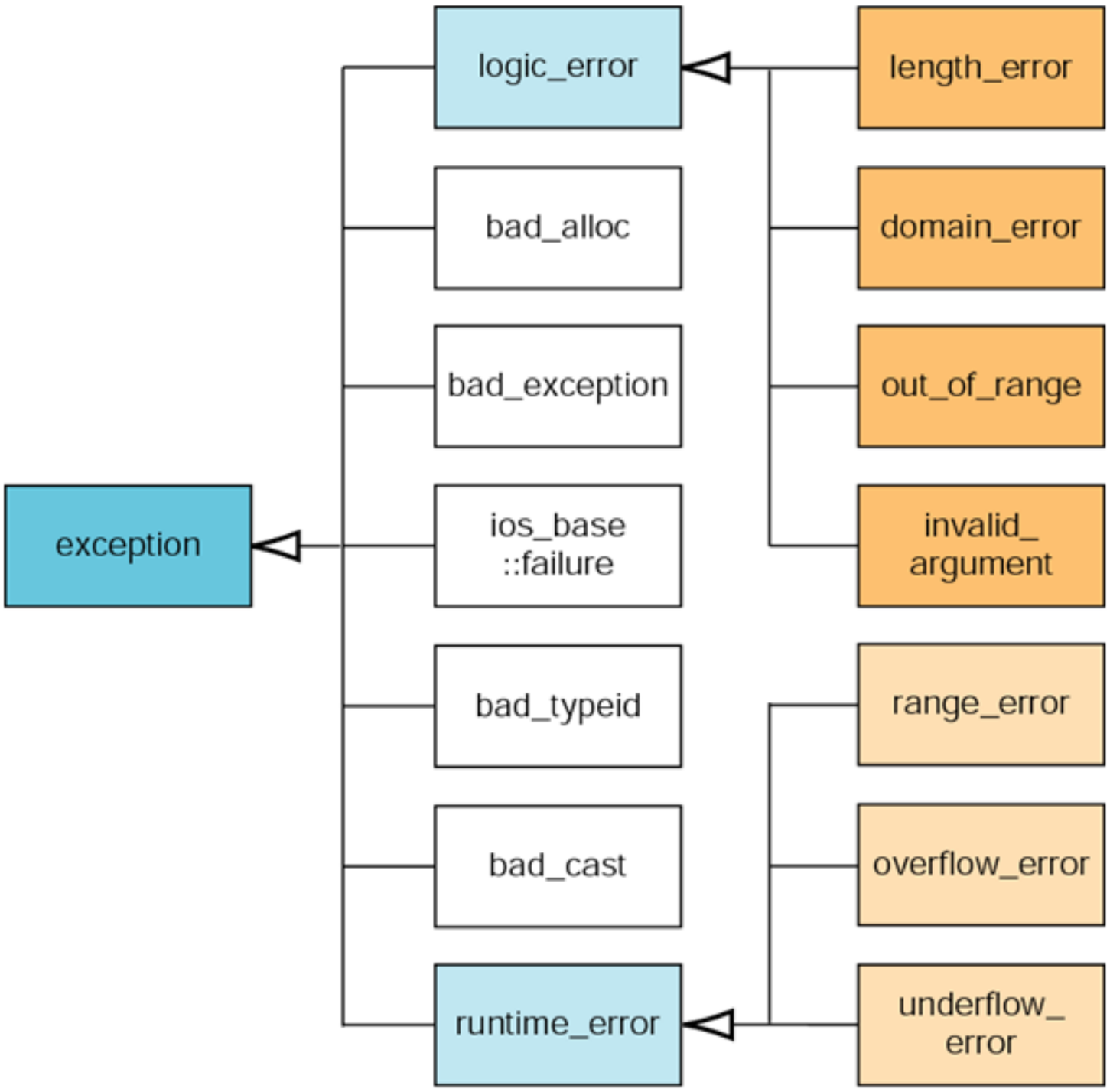
```
class A
{
public:
  A () {
    throw int ();
  }
};

A a;     // Implementation defined behaviour if exception is thrown (15.3/13)

int main ()
{
  try
  {
    // Exception for 'a' not caught here.
  }
  catch (int)
  {
  }
}
```

```
// Example 1(a): Constructor function try block
//
C::C()
try
  : A ( /*...*/ )   // optional initialization list
  , b_( /*...*/ )
{
}
catch( ... )
{
  //  We get here if either A::A() or B::B() throws.

  //  If A::A() succeeds and then B::B() throws, the
  //  language guarantees that A::~A() will be called
  //  to destroy the already-created A base subobject
  //  before control reaches this catch block.
}
```

Standard Exceptions

- So far we talked about programmer-defined exceptions
- Standard C++ functions can have their own exception handling logic
  - C++ defines an exception class *std::exception*, from which all standard exceptions are derived
- We can catch them if their calls are placed within a *try* statement

Standard exceptions



Standard Exceptions

Exception	Explanation
logic_error • domain_error • invalid_argument • length_error • out_of_range	Errors in the internal logic of the program • function argument is invalid • invalid argument in C++ standard function • object's length exceeds maximum allowable length • reference to array element out of range
bad_alloc	*new* cannot allocate memory
bad_exception	Exception doesn't match any *catch*
ios_base::failure	Error in processing external file
bad_typeid	Error in *typeid*
bad_cast	Failure in a dynamic cast
runtime_error • range_error • overflow_error • underflow_error	Errors beyond the scope of the program • error in standard template library (STL) container • overflow in STL container • underflow in STL container

what Method: Reason for Exceptions

- Standard exception class has a public virtual method called *what* that returns a C-string message explaining the error

```
cout << "Error: " << err.what() << endl;
```

- Since the *what* function is virtual, it is overridden in its derived classes using polymorphism
- Therefore, we can call it even if we access it through a base class

Standard error class example (1)

```
#include <iostream>
#include <string>
#include <iomanip>
#include <exception>
using namespace std;

int main ()
{
string s1 ("This is the string") ;

        cout << "Testing out_of_range exception\n";
        try
          {
          cout << "s1(125) contains: "
               << s1.at(125) << endl;
          } // try

        catch (exception& err)
          {

          cout << err.what()  << endl;
          } // catch
        cout << "End of exceptions tests\n";
        return 0;
}       // main
```

```
/*              Results:
Run 1:
                Testing out_of_range exception
                **basic_string::at index out of range
                End of exceptions tests
Run 2:  try block modified to valid index
                Testing out_of_range exception
                s1(5) contains: i
                End of exceptions tests
*/
```

Standard error class example (2)

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

int main ()
{
        cout << "Demonstrate memory allocation failure\n";
        try
          {
           double* Arr = new double [100000000000];
           cout << "Memory allocated successfully\n";
          } // try

        catch (exception& err)
          {
           cout << "**Error 100: Program  out of memory\n**"
                << err.what()    << endl ;
          } // catch

        cout << "End of exceptions tests\n";
        return 0;
}       // main
```

```
/*                Results
Demonstrate memory allocation failure
**Error 100: Program out of memory
**bad_alloc
End of exceptions tests
*/
```

Standard error class example (3)

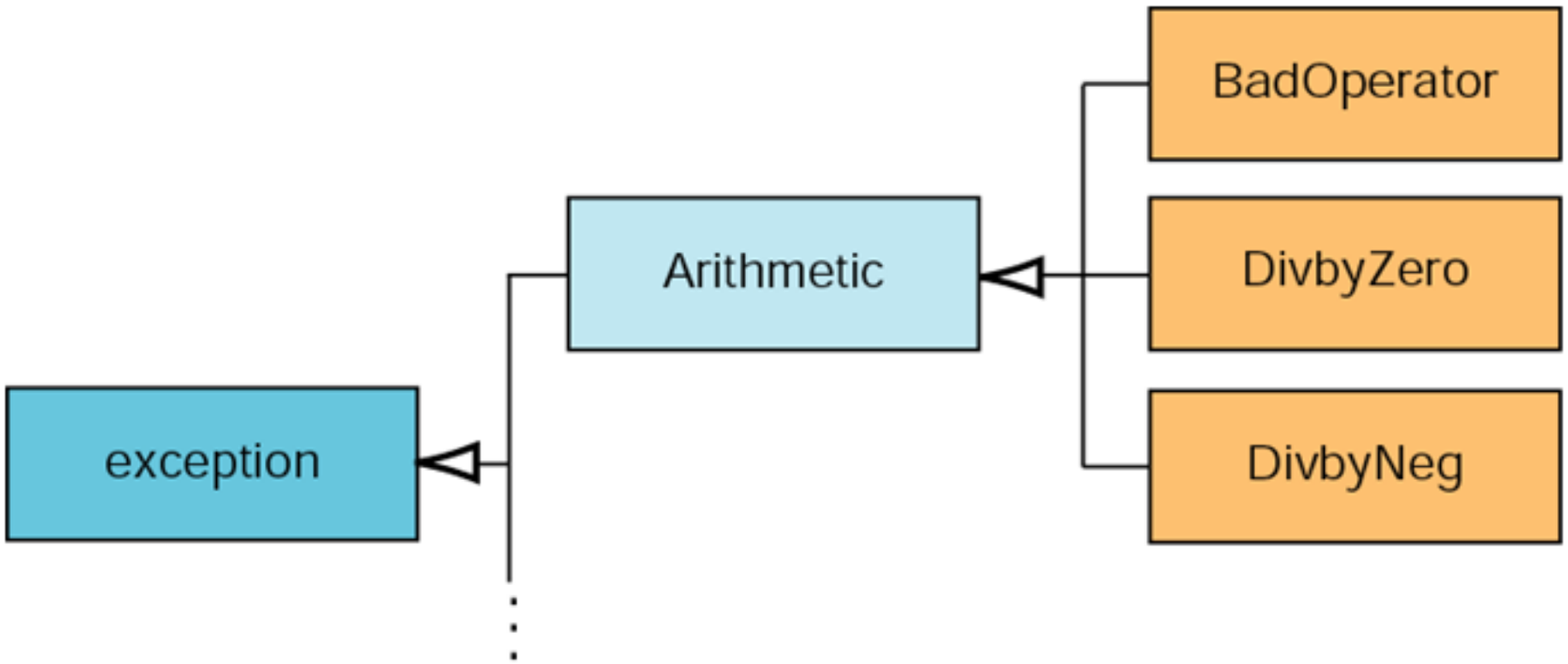
```
#include <iostream>
#include <fstream>
#include <stdexcept>
using namespace std;

int main ()
{
        ifstream noSuchFile;
        noSuchFile.exceptions(ios::badbit|ios::failbit);

        cout << "Testing open error\n";
        try
          {
          noSuchFile.open ("notThere", ios::in);
          cout << "File 'NotThere' successfully opened\a\n";
          } // try

        catch (exception& err)
          {
          cout << "Error opening file 'NotThere'\n**"
               << err.what() << endl;
          } // catch
        cout << "End of exceptions tests\n";
        return 0;
}       // main
```

```
/*              Results
Testing open error
Error opening file 'NotThere'
**ios_base failure in clear
End of exceptions tests
*/
```

Adding New Classes to the Standard Error Classes

- We can combine programmer-defined exception classes with the standard exception classes
- Use the standard exception class as a base class to the programmer-defined classes
- This allows us to use the standard exception class reference to catch all of our exceptions

Adding errors to standard error class (overloading virtual what function)



Example : Overloading what function

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

class Arithmetic: public exception
{
        public:
          virtual const char* what () const throw ()
                {return "**Error: type Arithmetic\n";}
};      // Arithmetic
class DivbyZero: public Arithmetic
{
        public:
          virtual const char* what () const throw ()
                {return "**Error: 100 divisor 0\n";}
};      // DivbyZero
class DivbyNeg: public Arithmetic
{
        public:
          virtual const char* what () const throw ()
                {return "**Error: 101 negative divisor\n";}
};      // DivbyZero
class BadOperator: public Arithmetic
{
        public:
          virtual const char* what () const throw ()
                {return "**Error: 102 invalid operator\n";}
};// BadOperator
```

Questions?