Exception Handling

Meng-Hsun Tsai
CSIE, NCKU

```
try {
        //   C++ statements
} catch (<exception type> <var>) {
        // exception handling codes
}
```

Introduction

- In this lecture, we introduce exception handling.
- An exception is an indication of a problem that occurs during a program's execution.
- The name "exception" implies that the problem occurs infrequently—if the "rule" is that a statement normally executes correctly, then the "exception to the rule" is that a problem occurs.
- Exception handling enables you to create applications that can resolve (or handle) exceptions.

Introduction (cont.)

- In many cases, handling an exception allows a program to continue executing as if no problem had been encountered.
- A more severe problem could prevent a program from continuing normal execution, instead requiring the program to notify the user of the problem before terminating in a controlled manner.
- The features presented in this lecture enable you to write robust and fault-tolerant programs that can deal with problems that may arise and continue executing or terminate gracefully.

Exception-Handling Overview

- Program logic frequently tests conditions that determine how program execution proceeds.
- Consider the following pseudocode:
    - *Perform a task*
    - *If the preceding task did not execute correctly*
      Perform error processing
    - *Perform next task*
    - *If the preceding task did not execute correctly*
      Perform error processing
      …
- In this pseudo code, we begin by performing a task. We then test whether that task executed correctly. If not, we perform error processing. Otherwise, we continue with the the next task.

IMSLab Since 2010

Exception-Handling Overview (cont.)

- Intermixing program logic with error-handling logic can make the program difficult to read, modify, maintain and debug—especially in large applications.
- If the potential problems occur infrequently, intermixing program logic and error-handling logic can degrade a program's performance, because the program must (potentially frequently) perform tests to determine whether the task executed correctly and the next task can be performed.

```
ofstream outFile("outfile", ios::out);
if(!outFile) {
    cerr << "Failed opening" << endl;
    exit(1);
}
```

Exception-Handling Overview (cont.)

- Exception handling enables you to remove error-handling code from the "main line" of the program's execution, which improves program clarity and enhances modifiability.
- You can decide to handle any exceptions you choose—all exceptions, all exceptions of a certain type or all exceptions of a group of related types (e.g., exception types that belong to an inheritance hierarchy).
- Such flexibility reduces the likelihood that errors will be overlooked and thereby makes a program more robust.

Example: Handling Exception for Midterm and Final Exams

```
class OverSleep {
public:
    OverSleep(string msg): err_msg(msg) {}
    string what() { return err_msg; }
private:
    string err_msg;
};
void midterm(int hour)
{
   cout << "midterm ing..." << endl;
   if(hour > 16)
      throw OverSleep("oversleep for midterm");
   // take exam
}
void final(int hour)
{
    cout << "final ing..." << endl;
    if(hour > 16)
      throw OverSleep("oversleep for final");
    // take exam
}
int main()
{
    try {
      midterm(17);
      final(17);
    } catch (OverSleep & overSleep) {
      cout << "exception occurred: "
            << overSleep.what();
      cout << "\nSee you next year !\n";
      // downdiao();
    }
    cout << "after try block" << endl;
    return 0;
}
```

```
midterm ing...
exception occurred: oversleep for midterm
See you next year !
after try block
```

Example: Handling an Attempt to Divide by Zero

- The purpose of this example is to show how to prevent a common arithmetic problem—division by zero.
- In C++, division by zero using integer arithmetic typically causes a program to terminate prematurely.
- In floating-point arithmetic, some C++ implementations allow division by zero, in which case positive or negative infinity is displayed as `INF` or `-INF`, respectively.

Example: Handling an Attempt to Divide by Zero (cont.)

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

int main()
{
    cout << 10/0 << endl;
    cout << "I am still alive!" << endl;
    return 0;
}
```

```
$ g++ -o div_by_zero div_by_zero.cpp
div_by_zero.cpp: In function `int main()':
div_by_zero.cpp:6: warning: division by zero
in `10 / 0'
$ ./div_by_zero
Floating point exception (core dumped)
```

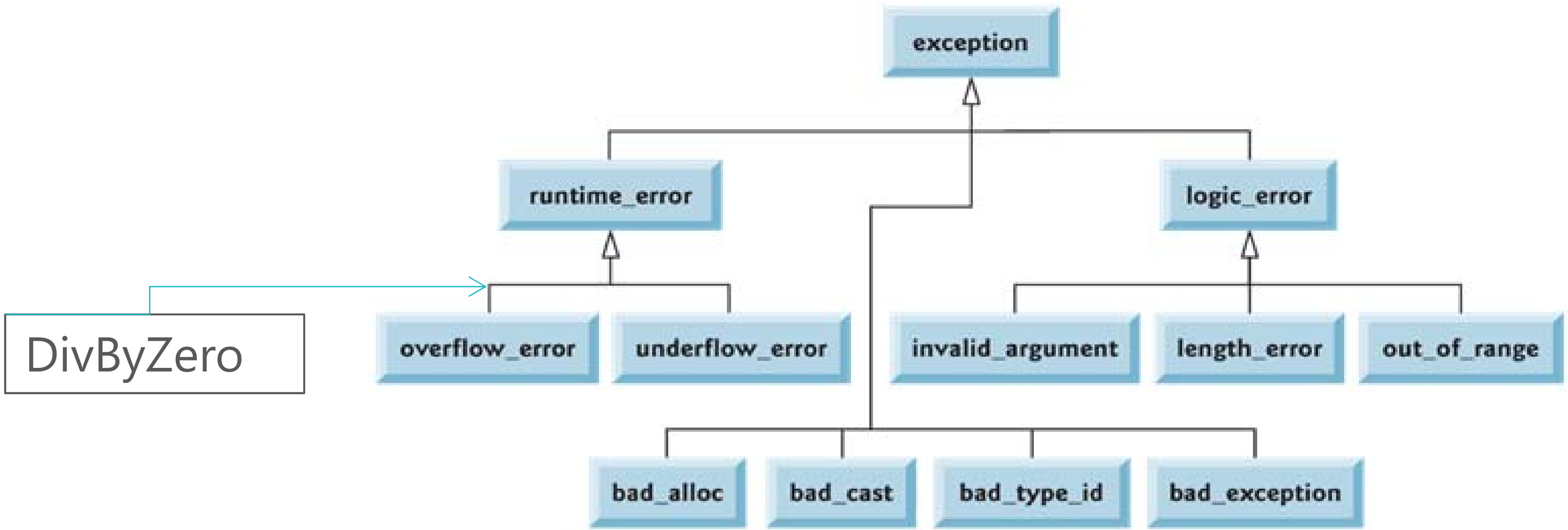
```
#include <iostream>
using namespace std;

int main()
{
    cout << 10.0/0 << endl;
    cout << "I am still alive!" << endl;
    return 0;
}
```

```
$ g++ -o div_by_zero div_by_zero.cpp
div_by_zero.cpp: In function `int main()':
div_by_zero.cpp:6: warning: division by zero
in `1.0e+1 / 0'
$ ./div_by_zero
inf
I am still alive!
```

Example: Handling an Attempt to Divide by Zero (cont.)

DivByZero.h

```
#include <stdexcept>
using namespace std;
class DivByZero: public runtime_error {
public:
   DivByZero(): runtime_error("divide by zero") {}
};
```



Example: Handling an Attempt to Divide by Zero (cont.)

```
#include <iostream>
#include “DivByZero.h"
using namespace std;
double quotient(int numerator, int denominator)
{
  if(denominator == 0)
     throw DivByZero();
  return static_cast<double>(numerator)
                          / denominator;
}
int main()
{
  int num1, num2;
  double result;
  cout << "Enter two integers: ";
  while (cin >> num1 >> num2)
  {
     try {
        result = quotient(num1, num2);
        cout << "quotient is " << result;
     } catch (DivByZero & divByZero) {
        cout << “Exception: ” <<
                 divByZero.what();
     }
     cout << "\nEnter two integers: ";
  }
}
```

```
Enter two integers: 100 7
quotient is 14.2857
Enter two integers: 100 0
Exception: divide by zero
Enter two integers:
```

Example: Handling an Attempt to Divide by Zero (cont.)

- In this example, we define a function named `quotient` that receives two integers input by the user and divides its first `int` parameter by its second `int` parameter.
- Before performing the division, the function casts the first `int` parameter's value to type `double`.
- Then, the second `int` parameter's value is promoted to type `double` for the calculation.
- So function `quotient` actually performs the division using two `double` values and returns a `double` result.

Example: Handling an Attempt to Divide by Zero (cont.)

- Although division by zero is allowed in floating-point arithmetic, for the purpose of this example we treat any attempt to divide by zero as an error.
- Thus, function `quotient` tests its second parameter to ensure that it isn't zero before allowing the division to proceed.
- If the second parameter is zero, the function uses an exception to indicate to the caller that a problem occurred.
- The caller (`main` in this example) can then process the exception and allow the user to type two new values before calling function `quotient` again.
- In this way, the program can continue to execute even after an improper value is entered, thus making the program more robust.

IMSLab Since 2010

Example: Handling an Attempt to Divide by Zero (cont.)

- The class `DivByZero` is defined as a derived class of Standard Library class `runtime_error` (defined in header file `<stdexcept>`).
- Class `runtime_error`—a derived class of Standard Library class `exception` (defined in header file `<exception>`)—is the C++ standard base class for representing runtime errors.
- Class `exception` is the standard C++ base class for all exceptions.

Example: Handling an Attempt to Divide by Zero (cont.)

- A typical exception class that derives from the `runtime_error` class defines only a constructor (e.g., line 5) 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, such as `DivByZero`, from the standard exception classes provided by C++.
  - Doing so allows you to use the `virtual` function `what` to obtain an appropriate error message.

Example: Handling an Attempt to Divide by Zero (cont.)

- The program uses exception handling to wrap code that might throw a “divide-by-zero” exception and to handle that exception.
- Function `quotient` divides its first parameter (`numerator`) by its second parameter (`denominator`).
- Assuming that the user does not specify `0` as the denominator for the division, function `quotient` returns the division result.
- However, if the user inputs `0` for the denominator, function `quotient` throws an exception.

Example: Handling an Attempt to Divide by Zero (cont.)

- C++ provides `try` blocks to enable exception handling.
- If an exception occurs, all of the enclosed statements should be skipped.

```
try {
    result = quotient(num1, num2);
    cout << "quotient is " << result;
} catch (DivByZero & divByZero) {
    cout << "Exception: " << divByZero.what();
}
```

```
Enter two integers: 100 7
quotient is 14.2857
Enter two integers: 100 0
Exception: divide by zero
Enter two integers:
```

Example: Handling an Attempt to Divide by Zero (cont.)

- Exceptions are processed by `catch` handlers (also called exception handlers), which catch and handle exceptions.
- At least one `catch` handler must immediately follow each `try` block.
- Each `catch` handler begins with the keyword `catch` and specifies in parentheses an exception parameter that represents the type of exception the `catch` handler can process (`DivByZero` in this case).
- When an exception occurs in a `try` block, the `catch` handler that executes is the one whose type matches the type of the exception that occurred (i.e., the type in the `catch` block matches the thrown exception type exactly or is a base class of it).

Example: Handling an Attempt to Divide by Zero (cont.)

- If an exception parameter includes an optional parameter name, the `catch` handler can use that parameter name to interact with the caught exception in the body of the `catch` handler, which is delimited by braces (`{` and `}`).
- A `catch` handler typically reports the error to the user, logs it to a file, terminates the program gracefully or tries an alternate strategy to accomplish the failed task.
- In this example, the `catch` handler simply reports that the user attempted to divide by zero. Then the program prompts the user to enter two new integer values.

Example: Handling an Attempt to Divide by Zero (cont.)

- It's a syntax error to place code between a `try` block and its corresponding `catch` handlers or between its `catch` handlers.
- Each `catch` handler can have only a single parameter—specifying a comma-separated list of exception parameters is a syntax error.
- It's a logic error to catch the same type in two different `catch` handlers following a single `try` block.

Example: Handling an Attempt to Divide by Zero (cont.)

- If an exception occurs as the result of a statement in a `try` block, the `try` block expires (i.e., terminates immediately).
- Next, the program searches for the first `catch` handler that can process the type of exception that occurred.
- The program locates the matching `catch` by comparing the thrown exception's type to each `catch`'s exception-parameter type until the program finds a match.
- A match occurs if the types are identical or if the thrown exception's type is a derived class of the exception-parameter type.
- When a match occurs, the code contained in the matching `catch` handler executes.

Example: Handling an Attempt to Divide by Zero (cont.)

- When a `catch` handler finishes processing by reaching its closing right brace (`}`), the exception is considered handled and the local variables defined within the `catch` handler (including the `catch` parameter) go out of scope.
- Program control does not return to the point at which the exception occurred (known as the throw point), because the `try` block has expired.
- Rather, control resumes with the first statement after the last `catch` handler following the `try` block.
- This is known as the termination model of exception handling.
- As with any other block of code, when a `try` block terminates, local variables defined in the block go out of scope.

Example: Handling an Attempt to Divide by Zero (cont.)

- If the `try` block completes its execution successfully (i.e., no exceptions occur in the `try` block), then the program ignores the `catch` handlers and program control continues with the first statement after the last `catch` following that `try` block.

- If an exception that occurs in a `try` block has no matching `catch` handler, or if an exception occurs in a statement that is not in a `try` block, the function that contains the statement terminates immediately, and the program attempts to locate an enclosing `try` block in the calling function.

- This process is called stack unwinding.

Example: Handling an Attempt to Divide by Zero (cont.)

- As part of throwing an exception, the `throw` operand is created and used to initialize the parameter in the `catch` handler, which we discuss momentarily.
- Central characteristic of exception handling: A function should throw an exception before the error has an opportunity to occur.
- In this program, the `catch` handler specifies that it catches `DivByZero` objects—this type matches the object type thrown in function `quotient`.
- Actually, the `catch` handler catches a reference to the `DivByZero` object created by function `quotient`'s `throw` statement.

When to Use Exception Handling

- Exception handling is designed to process synchronous errors, which occur when a statement executes.
- Common examples of these errors are out-of-range array subscripts, arithmetic overflow (i.e., a value outside the representable range of values), division by zero, invalid function parameters and unsuccessful memory allocation (due to lack of memory).
- Exception handling is not designed to process errors associated with asynchronous events (e.g., disk I/O completions, network message arrivals, mouse clicks and keystrokes), which occur in parallel with, and independent of, the program's flow of control.

When to Use Exception Handling (cont.)

- Incorporate your exception-handling strategy into your system from inception. Including effective exception handling after a system has been implemented can be difficult.
- When no exceptions occur, exception-handling code incurs little or no performance penalty. Thus, programs that implement exception handling operate more efficiently than programs that intermix error-handling code with program logic.

When to Use Exception Handling (cont.)

- The exception-handling mechanism also is useful for processing problems that occur when a program interacts with software elements, such as member functions, constructors, destructors and classes.
- Rather than handling problems internally, such software elements often use exceptions to notify programs when problems occur.
- This enables you to implement customized error handling for each application.

When to Use Exception Handling (cont.)

- Complex applications normally consist of predefined software components and application-specific components that use the predefined components.
- When a predefined component encounters a problem, that component needs a mechanism to communicate the problem to the application-specific component—the predefined component cannot know in advance how each application processes a problem that occurs.

Rethrowing an Exception

- It's possible that an exception handler, upon receiving an exception, might decide either that it cannot process that exception or that it can process the exception only partially.
- In such cases, the exception handler can defer the exception handling (or perhaps a portion of it) to another exception handler.
- In either case, you achieve this by rethrowing the exception via the statement
  - `throw;`
- The next enclosing `try` block detects the rethrown exception, which a `catch` handler listed after that enclosing `try` block attempts to handle.

Rethrowing an Exception (cont.)

```
#include <iostream>
#include <exception>
using namespace std;
void throwException()
{
   try {
      cout << "throw exception\n";
      throw exception();
   } catch (exception &) {
      cout << "exception caught in
                 throwException()\n";
      throw;
   }
   cout << "this line should not appear\n";
}
int main()
{
   try {
      cout << "in main()\n";
      throwException();
      cout << "after throwException()\n";
   } catch (exception &){
      cout << "exception caught in main()\n";
   }
   cout << "after catch in main()\n";
}
```

```
in main()
throw exception
exception caught in throwException()
exception caught in main()
after catch in main()
```

Rethrowing an Exception (cont.)

- Since we do not use the exception parameters in the `catch` handlers of this example, we omit the exception parameter names and specify only the type of exception to catch.

```
9     } catch (exception &) {
```

```
21    } catch (exception &){
```

Error: Rethrowing an Empty Exception

- Executing an empty `throw` statement outside a `catch` handler calls function `terminate`, which abandons exception processing and terminates the program immediately.

```
$ cat throw.cpp
#include <iostream>
using namespace std;
int main()
{
	throw;
	cout << "after throw" << endl;
	return 0;
}
$ g++ -o throw throw.cpp
$ ./throw.exe
Aborted (core dumped)
```

Exception Specifications

- An optional exception specification (also called a `throw` list) enumerates a list of exceptions that a function can throw.

- For example, consider the function declaration

    ```
    int someFunction( double value )
        throw (ExceptionA, ExceptionB, ExceptionC)
    {
        // function body
    }
    ```

- Indicates that function `someFunction` can throw exceptions of types `ExceptionA`, `ExceptionB` and `ExceptionC`.

Exception Specifications (cont.)

- A function can throw only exceptions of the types indicated by the specification or exceptions of any type derived from these types.
- If the function `throw`s an exception that does not belong to a specified type, the exception-handling mechanism calls function `unexpected`, which terminates the program.
- A function that does not provide an exception specification can `throw` any exception.
- Placing `throw()`—an empty exception specification—after a function's parameter list states that the function does not `throw` exceptions.

Exception Specifications (cont.)

- If the function attempts to `throw` an exception, function `unexpected` is invoked.
- [*Note: Some compilers ignore exception specifications.]*
- The compiler will not generate a compilation error if a function contains a throw expression for an exception not listed in the function's exception specification. An error occurs only when that function attempts to throw that exception at execution time.
- To avoid surprises at execution time, carefully check your code to ensure that functions do not throw exceptions not listed in their exception specifications.

Processing Unexpected Exceptions

- Function `unexpected` calls the function registered with function `set_unexpected` (defined in header file `<exception>`).
- If no function has been registered in this manner, function `terminate` is called by default.
- Cases in which function `terminate` is called include:
  - the exception mechanism cannot find a matching `catch` for a thrown exception
  - a destructor attempts to `throw` an exception during stack unwinding
  - an attempt is made to rethrow an exception when there is no exception currently being handled
  - a call to function `unexpected` defaults to calling function `terminate`

Processing Unexpected Exceptions (cont.)

- Function `set_terminate` can specify the function to invoke when `terminate` is called.
- Otherwise, `terminate` calls `abort`, which terminates the program without calling the destructors of any remaining objects of automatic or static storage class.
- Aborting a program component due to an uncaught exception could leave a resource—such as a file stream or an I/O device—in a state in which other programs are unable to acquire the resource. This is known as a resource leak.

Processing Unexpected Exceptions (cont.)

- Function `set_terminate` and function `set_unexpected` each return a pointer to the last function called by `terminate` and `unexpected`, respectively (0, the first time each is called).
- This enables you to save the function pointer so it can be restored later.
- Functions `set_terminate` and `set_unexpected` take as arguments pointers to functions with `void` return types and no arguments.
- If the last action of a programmer-defined termination function is not to exit a program, function `abort` will be called to end program execution after the other statements of the programmer-defined termination function are executed.

Processing Unexpected Exceptions (cont.)

```
void midterm throw (Typhoon, Earthquake, Wedding) { … }
void unexpected_func() { downdiao(); }
int main()
{
    set_unexpected (unexpected_func) ;
    try {
      midterm();
    } catch (Typhoon & typhoon) { …
    } catch (Earthquake & earthquake) { …
    } catch (Wedding & wedding) { …
    }
    return 0;
}
```

Stack Unwinding

- When an exception is thrown but not caught in a particular scope, the function call stack is “unwound,” and an attempt is made to `catch` the exception in the next outer `try`…`catch` block.
- Unwinding the function call stack means that the function in which the exception was not caught terminates, all local variables in that function are destroyed and control returns to the statement that originally invoked that function.
- If no `catch` handler ever catches this exception, function `terminate` is called to terminate the program.

Stack Unwinding (cont.)

```
#include <iostream>
#include <stdexcept>
using namespace std;
void fun3() throw(runtime_error)
{
   cout << "in fun3()\n";
   throw runtime_error("runtime_error in fun3()");
}
void fun2() throw (runtime_error)
{
   cout << "in fun2()\n";
   fun3();
}
void fun1() throw(runtime_error)
{
   cout << "in fun1()\n";
   fun2();
}
int main()
{
   try {
      cout << "in main()\n";
      fun1();
   } catch(runtime_error & err) {
      cout << "exception: " <<
                 err.what() << endl;
   }
}
```

```
in main()
in fun1()
in fun2()
in fun3()
exception: runtime_error in fun3()
```

main() -> fun1() -> fun2() -> fun3()

try / catch

throw

Stack Unwinding (cont.)

- Line 7 of `fun3` throws a `runtime_error` object.
- However, because no `try` block encloses the `throw` statement in line 7, stack unwinding occurs—`fun3` terminates at line 7, then returns control to the statement in `fun2` that invoked `fun3` (i.e., line 12).
- Because no `try` block encloses line 12, stack unwinding occurs again—`fun2` terminates at line 12 and returns control to the statement in `fun1` that invoked `fun2` (i.e., line 17).
- Because no `try` block encloses line 17, stack unwinding occurs one more time—`fun1` terminates at line 17 and returns control to the statement in `main` that invoked `fun1` (i.e., line 23).
- The `try` block of lines 21–23 encloses this statement, so the first matching `catch` handler located after this `try` block (line 24–26) catches and processes the exception.

Processing new Failures

- The C++ standard specifies that, when operator `new` fails, it `throw`s a `bad_alloc` exception (defined in header file `<new>`).
- In this section, we present two examples of `new` failing.
  - The first uses the version of `new` that `throw`s a `bad_alloc` exception when `new` fails.
  - The second uses function `set_new_handler` to handle `new` failures.

Processing new Failures (cont.)

```
#include <iostream>
#include <new>
using namespace std;
int main()
{
   double *ptr[50];
   try {
      for(int i = 0; i<50; ++i)
      {
         ptr[i] = new double[50000000];
         cout << "ptr[" << i << "] new success\n";
      }
   } catch(bad_alloc &memoryAlloc){
      cerr << "Exception: " <<
                  memoryAlloc.what() << endl;
   }
}
```

```
ptr[0] new success
ptr[1] new success
ptr[2] new success
ptr[3] new success
ptr[4] new success
ptr[5] new success
ptr[6] new success
Exception: std::bad_alloc
```

```
Output of top:
25580 tsaimh      1  50    0  2678M  1432K ttyin   1   0:00  0.00% new_failure
```

Processing new Failures (cont.)

- The program demonstrates `new` throwing `bad_alloc` on failure to allocate the requested memory.
- The `for` statement (lines 8–12) inside the `try` block should loop 50 times and, on each pass, allocate an array of 50,000,000 `double` values.
- If `new` fails and throws a `bad_alloc` exception, the loop terminates, and the program continues in line 13, where the `catch` handler catches and processes the exception.
- Line 14 print the message `"Exception:"` followed by the message returned from the base-class-`exception` version of function `what` (i.e., an implementation-defined exception-specific message, such as `"Allocation Failure"` in Microsoft Visual C++).

Processing new Failures (cont.)

- The output shows that the program performed only six iterations of the loop before `new` failed and threw the `bad_alloc` exception.
- Your output might differ based on the physical memory, disk space available for virtual memory on your system and the compiler you are using.

Processing new Failures (cont.)

- In old versions of C++, operator `new` returned 0 when it failed to allocate memory.
- The C++ standard specifies that standard-compliant compilers can continue to use a version of `new` that returns `0` upon failure.
- For this purpose, header file `<new>` defines object `nothrow` (of type `nothrow_t`), which is used as follows:
  - `double *ptr = new( nothrow ) double[ 50000000 ];`
- The preceding statement uses the version of `new` that does not throw `bad_alloc` exceptions (i.e., `nothrow`) to allocate an array of 50,000,000 `double`s.
- To make programs more robust, use the version of `new` that throws `bad_alloc` exceptions on failure.

IMSLab Since 2010

Using *new* Handler to Process *New* Failure

```cpp
#include <iostream>
#include <new>
#include <cstdlib>
using namespace std;
void customNewHandler()
{
   cerr << "in customNewHandler()\n";
   abort();
}
int main()
{
   double *ptr[50];
   set_new_handler(customNewHandler);
   for(int i=0; i<50; ++i)
   {
      ptr[i] = new double[50000000];
      cout << "ptr[" << i << "] new success\n";
   }
}
```

```
ptr[0] new success
ptr[1] new success
ptr[2] new success
ptr[3] new success
ptr[4] new success
ptr[5] new success
ptr[6] new success
in customNewHandler()

Abort (core dumped)
```

Using *new* Handler to Process *New* Failure (cont.)

- Function `set_new_handler` (prototyped in `<new>`) takes as its argument a pointer to a function that takes no arguments and returns `void`.
  - This pointer points to the function that will be called if `new` fails.
  - This provides you with a uniform approach to handling all `new` failures, regardless of where a failure occurs in the program.
- Once `set_new_handler` registers a `new` handler in the program, operator `new` does not throw `bad_alloc` on failure; rather, it defers the error handling to the `new`-handler function.

Using *new* Handler to Process *New* Failure (cont.)

- If `new` fails to allocate memory and `set_new_handler` did not register a `new`-handler function, `new` throws a `bad_alloc` exception.
- If `new` fails to allocate memory and a `new`-handler function has been registered, the `new`-handler function is called.
- Function `customNewHandler` (lines 5–9) prints an error message, then calls `abort` to terminate the program.
- The output shows that the loop iterated six times before `new` failed and invoked function `customNewHandler`.

Using *new* Handler to Process *New* Failure (cont.)

- The C++ standard specifies that the `new`-handler function should perform one of the following tasks:
  - Make more memory available by deleting other dynamically allocated memory (or telling the user to close other applications) and return to operator `new` to attempt to allocate memory again.
  - Throw an exception of type `bad_alloc`.
  - Call function `abort` or `exit` (both found in header file `<cstdlib>`) to terminate the program.

Using *new* Handler to Process *New* Failure (cont.)

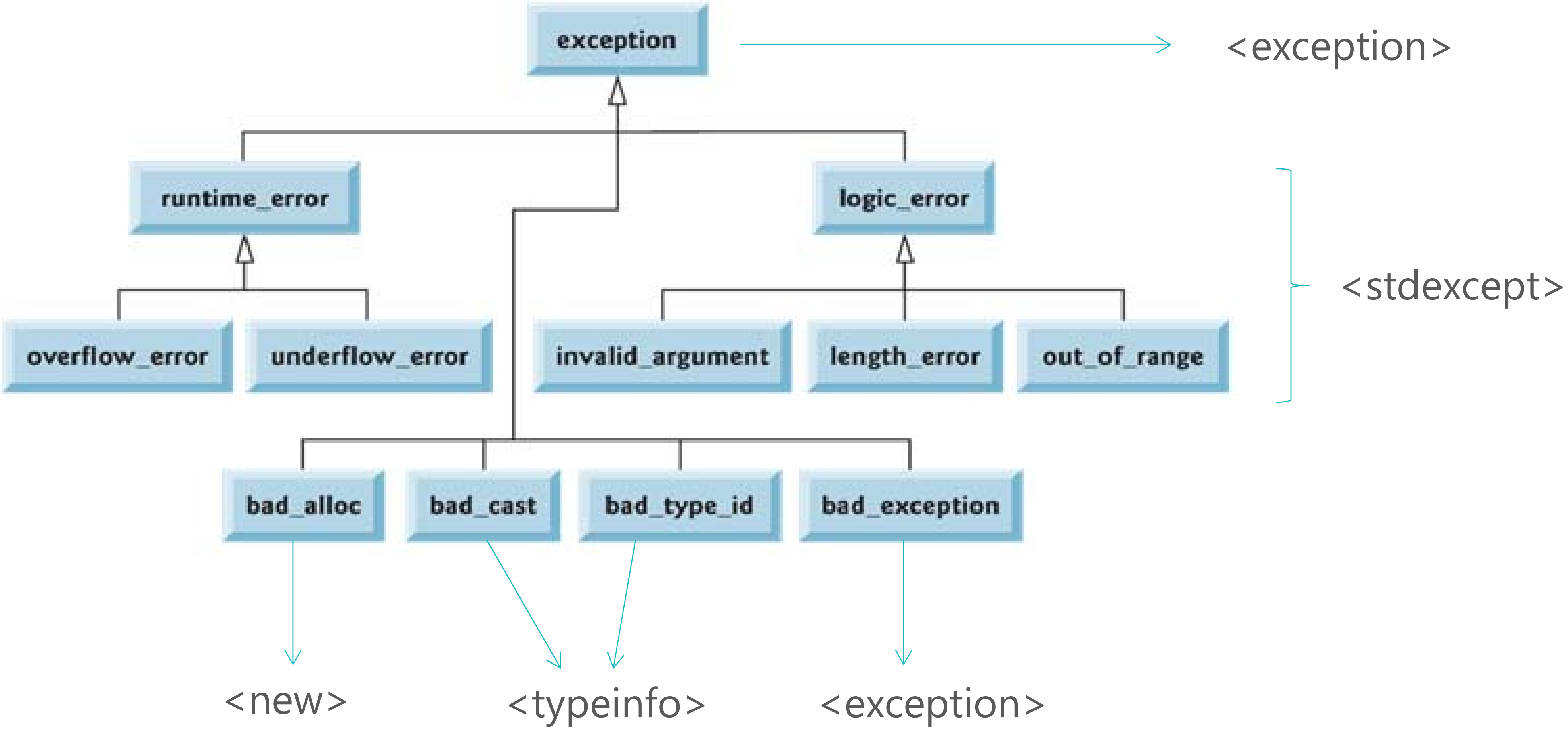
```
double * double_ptr [50];
int j = 0;
void new_hand()
{
   cout << "new_hand(): delete double_ptr["<< j
          << ", "  << j+1 << "]" << endl;
   delete [] double_ptr[j++];
   delete [] double_ptr[j++];
}
int main()
{
  set_new_handler(new_hand);
  for(int i = 0; i< 10; i++)
  {
     cout << "for loop: new double_ptr[" << i
          << "]" << endl;
     double_ptr[i] = new double[50000000];
  }
  return 0;
}
```

```
for loop: new double_ptr[0]
for loop: new double_ptr[1]
for loop: new double_ptr[2]
for loop: new double_ptr[3]
for loop: new double_ptr[4]
new_hand(): delete double_ptr[0, 1]
for loop: new double_ptr[5]
for loop: new double_ptr[6]
new_hand(): delete double_ptr[2, 3]
for loop: new double_ptr[7]
for loop: new double_ptr[8]
new_hand(): delete double_ptr[4, 5]
for loop: new double_ptr[9]
```

Standard Library Exception Hierarchy

- Experience has shown that exceptions fall nicely into a number of categories.
- The C++ Standard Library includes a hierarchy of exception classes, some of which are shown in the next slide.
- As we previously discussed, this hierarchy is headed by base-class `exception` (defined in header file `<exception>`), which contains `virtual` function `what`, which derived classes can override to issue appropriate error messages.

Standard Library Exception Hierarchy (cont.)



Standard Library Exception Hierarchy (cont.)

/usr/include/c++/4.2/exception

```
 class exception  {
  public:
     exception() throw() { }
     virtual ~exception() throw();
     virtual const char* what() const throw();
};
class bad_exception : public exception  {
  public:
     bad_exception() throw() { }
     virtual ~bad_exception() throw();
};
```

Standard Library Exception Hierarchy (cont.)

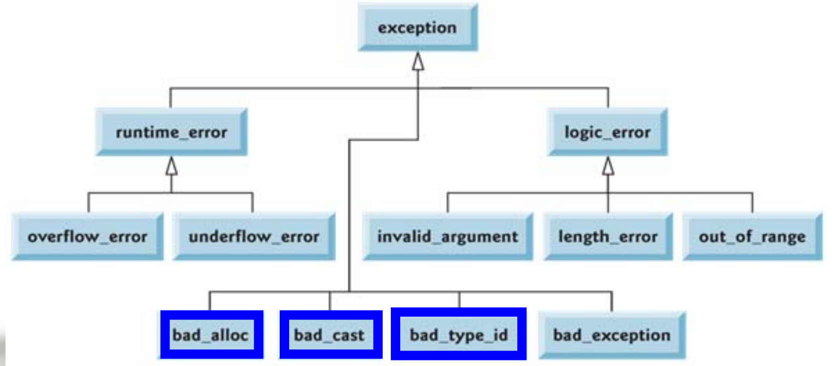
/usr/include/c++/4.2/stdexcept

```
class logic_error : public exception  {
    string _M_msg;
  public:
    /** Takes a character string describing the error.  */
    explicit  logic_error(const string&  __arg);
    virtual    ~logic_error() throw();

   /** Returns a C-style character string describing the general cause
     * of the current error (the same string passed to the ctor).  */
    virtual const char*  what() const throw();
};
```

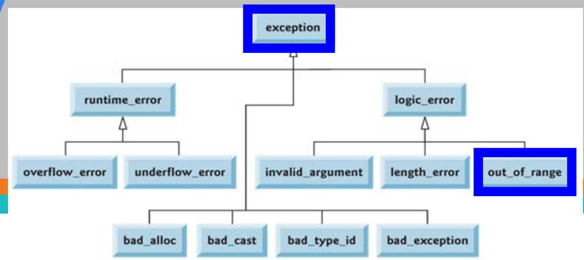
Standard Library Exception Hierarchy (cont.)

- Immediate derived classes of base-class `exception` include `runtime_error` and `logic_error` (both defined in header `<stdexcept>`), each of which has several derived classes.

- Also derived from `exception` are the exceptions thrown by C++ operators—for example, `bad_alloc` is thrown by `new`, `bad_cast` is thrown by `dynamic_cast` and `bad_typeid` is thrown by `typeid`.



Standard Library Exception Hierarchy (cont.)



- Placing a catch handler that catches a base-class object before a catch that catches an object of a class derived from that base class is a logic error. The derived-class catch will never execute.

```
string x = “NCKU”;
try {
    cout << x.substr(10,3) << endl;
} catch (out_of_range) {
    cerr << "out_of_range" << endl;
} catch (exception) {
    cerr << "exception caught" << endl;
}
```

```
string x = “NCKU”;
try {
    cout << x.substr(10,3) << endl;
} catch (exception) {
    cerr << "exception caught" << endl;
} catch (out_of_range) {
    cerr << "out_of_range" << endl;
}
```

```
exception2.cpp: In function `int main()':
exception2.cpp:12: warning: exception of type `std::out_of_range' will be caught
exception2.cpp:10: warning:    by earlier handler for `std::exception'
```

Standard Library Exception Hierarchy (cont.)

- Class `logic_error` is the base class of several standard exception classes that indicate errors in program logic.
  - For example, class `invalid_argument` indicates that an invalid argument was passed to a function.
  - Proper coding can, of course, prevent invalid arguments from reaching a function.
- Class `length_error` indicates that a length larger than the maximum size allowed for the object being manipulated was used for that object.
- Class `out_of_range` indicates that a value, such as a subscript into an array, exceeded its allowed range of values.

IMSLab Since 2010

Standard Library Exception Hierarchy (cont.)

- Class `runtime_error` is the base class of several other standard exception classes that indicate execution-time errors.
  - For example, class `overflow_error` describes an arithmetic overflow error (i.e., the result of an arithmetic operation is larger than the largest number that can be stored in the computer) and class `underflow_error` describes an arithmetic underflow error (i.e., the result of an arithmetic operation is smaller than the smallest number that can be stored in the computer).

Attempt to Catch All Exceptions

- Exception classes need not be derived from class `exception`, so catching type `exception` is not guaranteed to `catch` all exceptions a program could encounter.
- To `catch` all exceptions potentially thrown in a try block, use `catch(...)`. One weakness with catching exceptions in this way is that the type of the caught exception is unknown at compile time. Another weakness is that, without a named parameter, there is no way to refer to the exception object inside the exception handler.

```
string x = "NCKU";
  try {
     cout << x.substr(7,2);
  } catch (...) {
     cerr << "everyting caught" << endl;
  }
```