UCLA PIC 10 B

Problem Solving using C++ Programming

- **Instructor**: **Ivo Dinov**, Asst. Prof. in Mathematics, Neurology, Statistics
- **Teaching Assistant:** Suzanne Nezzar, Mathematics

University of California, Los Angeles, Summer 2001

http://www.math.ucla.edu/~dinov/10b.1.011/

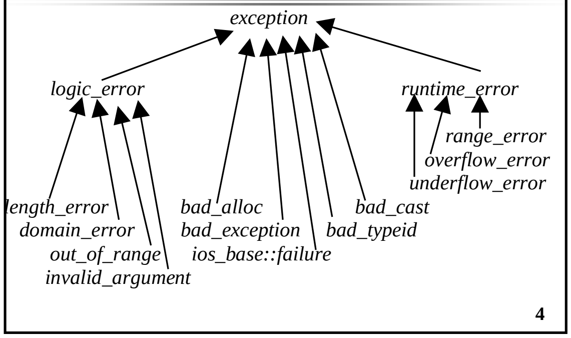
Chapter 16

Exception Handling

16 Exception Handling

- Exception Handling Basics
  - A Toy Example of Exception Handling
  - Defining Your Own Exception Classes
  - Multiple Throws and Catches
  - Throwing an Exception in a Function
- Programming Techniques for Exception Handling
  - When to Throw an Exception
  - Exception Class Hierarchies
  - Testing for Available Memory
  - Rethrowing an Exception

16 Exception Handling



16 Exception Handling

Introduction (1 of 2)

- One way to write programs is to assume nothing unusual will happen and no errors will occur.
- This is, of course, grossly optimistic.
- Once the program is running (correctly) where everything goes as expected, then code is added to account for unusual cases.
- Exception handling is commonly used to handle error cases, but a better way is to view exception handling as a way to manage exceptional situations.
- If a program correctly handles an "error", then it is no longer an error.

Exception Handling

Introduction (2 of 2)

- Typically, exception handling deals with functions that have special cases that are best handled in a way specific to the use of the function.
- For some invocations a function should end, other invocations require another action .
- Such a function can be defined to throw an exception if a special case occurs, and the exception mechanism allows the special case to be handled *outside* the function.
- C++ provides a mechanism that, when an exceptional situation has occurred, program control is transferred to another code segment, and to send information about the situation to that code.
- This mechanism is called throwing an exception.
- There is a code segment that receives control from and information about the situation and manages the exceptional situation is called handling the exception.

16.1 Exception Handling Basics

Some points to ponder:

- Exception handling should be used sparingly.
- Exceptional situations are not necessarily errors.
- Exceptional handling examples are usually more involved than the following examples.

A Toy Example of Exception Handling(1 of 14)

- This toy example introduces exception handling ideas and C++ exception handling syntax.
- The initial code fragment computes a ratio of donuts to milk.
- A limitless supply of milk is assumed.

```
cin >> donuts; // number of donuts, int
cin >> milk;   // number of glasses, int
dpg = donuts/double(milk);
cout << "There are " << dpg << " donuts per glass of milk.\n";
```

- If there is no milk, this code divides by zero, which is an error.
- We can add a test to protect against such a situation.

A Toy Example of Exception Handling(2 of 14)

- A complete program to manage this is in Display 16.1.
- This program does not use exception handling.
- In Display 16.2 we rewrite the program using C++ exception handling.
- The program is not made simpler by use of exceptions, but the part in the block after the keyword *try* and before the keyword *catch* is cleaner.
- This hints at the advantage of using exceptions.

A Toy Example of Exception Handling (3 of 14)

- Display 16.1 has a large *if-else* statement the manage the zero divide.
- The new program has the smaller *if* statement:

```
if (milk <= 0)
   throw donuts;
```

- This *if* statement says if there is no milk, an exceptional situation exists, do something to manage it.
- The normal situation is managed by code following *try* and code following *catch* manages the exceptional circumstances.
- We have separated the normal case from the exceptional case.

Display 16.1 Handling a Special Case without Exception Handling

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

int main( )
{   int donuts, milk;
    double dpg;
    cout << "Enter number of donuts:\n";
    cin >> donuts;
    cout << "Enter number of glasses of milk:\n";
    cin >> milk;
    if (milk <= 0)
      cout << donuts << " donuts, and No Milk!\n"  << "Go buy some milk.\n";
    else
    {   dpg = donuts/double(milk);
        cout << donuts << " donuts.\n" << milk << " glasses of milk.\n"
                 << "You have " << dpg  << " donuts for each glass of milk.\n";
    }
    cout << "End of program.\n";
    return 0;
}
```

Display 16.2 Same thing Using Exception Handling

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

int main( )
{   int donuts, milk;
    double dpg;
    try
    {   cout << "Enter number of donuts:\n";
        cin >> donuts;
        cout << "Enter number of glasses of milk:\n";
        cin >> milk;
        if (milk <= 0)
           throw donuts;
        dpg = donuts/double(milk);
        cout << donuts << " donuts.\n"  << milk << " glasses of milk.\n"
                << "You have " << dpg << " donuts for each glass of milk.\n";
    }
    catch(int e)
    {   cout << e << " donuts, and No Milk!\n"  << "Go buy some milk.\n";  }
    cout << "End of program.\n";
    return 0;
}
```

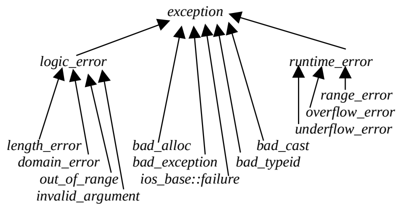
Review, Tuesday, Aug. 07, 2001
PIC 10 B

- **Virtual functions:**

```
class Pet {                 virtual void print( );
                            string name; };
class Dog : public Pet{     virtual void print( );
                            string breed;   };
Dog vdog;
Pet vpet;
vdog.name = "Tiny"; vdog.breed = "Great Dane"; vpet = vdog;
cout << vpet.breed;  // Error: class Pet has no breed.
However,   Pet *ppet;
           Dog *pdog;
pdog=new Dog; pdog->name="Tiny"; pdog->breed="Great Dane";
ppet = pdog;
ppet→print();           // (*ppet).print();  WORKS
    // because print() is a  virtual member of Pet and Dog
```

Review, Tuesday, Aug. 07, 2001
PIC 10 B

- **Started with Exceptions, Ch. 16**



exception

logic_error

runtime_error

range_error
overflow_error
underflow_error

length_error
domain_error
out_of_range
invalid_argument

bad_alloc
bad_exception
ios_base::failure

bad_cast
bad_typeid

```
try  {  if (milk <= 0)      throw donuts;
    dpg = donuts/double(milk);   }
catch(int e)
{   cout << e << " donuts, and No Milk!\n"  << "Go buy some milk.\n";  }
```

A Toy Example of Exception Handling (4 of 14)

- If this were all there is to exception handling, we wouldn't have bought much.
- The C++ exception handling mechanism consists of `try-throw-catch` triple.
- A try block must be followed by a catch block.
- A try block has the syntax:

```
try   {    Some_code;  }
```

- The try block contains the code for the algorithm, to be used when things go smoothly.
- The block is called a *try*-block because we aren't sure things will go smoothly, but we want to "give it a try".

A Toy Example of Exception Handling (5 of 14)

- If something does go wrong we need to throw an exception to indicate that something has gone wrong.
- We add a throw statement controlled by some conditional:

```
try
{  Some_code_to_try;
   Possibly_throw_an_exception;
   More_code;
}
```

- In the next slide we copy the try-block from Display 16.2.

A Toy Example of Exception Handling (6 of 14)

- The try block from Display 16.2 is:

```
try {  cout << "Enter number of donuts:\n";
         cin >> donuts;
       cout << "Enter number of glasses of milk:\n";
       cin >> milk;
       if (milk <= 0)     throw donuts;
       dpg = donuts/double(milk);
       cout << donuts << " donuts.\n"  << milk << " glasses of milk.\n"
           << "You have " << dpg << " donuts for each glass of milk.\n";
}
```

- The line, `throw donuts;` throws the int value `donuts (exception)`.
- This value is called an exception.
- Executing a `throw` statement is called throwing an exception.
- Values of any type, including class type, can be thrown.

throw-Statement

Syntax:

```
throw Expression_for_Value_to_be_Thrown;
```

When the `throw` statement is executed, the execution of the enclosing `try`-block is followed by a suitable `catch`-block, then flow of control is transferred to the `catch`-block. A `throw`-statement is almost always embedded in a branching statement, such as an `if`-statement. The value thrown can be of any type.

Example:

```
if (milk <= 0)
  throw donuts;
```

catch-Block Parameter

The catch-block parameter is an identifier in the heading of a catch-block that serves as a place holder for an exception (a value) that might be thrown. When a (suitable) value is thrown in the preceding try-block, that value is plugged in for the catch-block parameter. You can use any legal (non-reserved word) identifier for a catch-block parameter.

Example:

```
catch(int e)
{   cout << e << " donuts, and no milk!\n"
       << "Go buy some milk.\n";
}
```

Here, e is the catch block parameter.

A Toy Example of Exception Handling (7 of 14)

- **The word throw suggests that something goes from one place to another.**
- **In C++ flow of control is passed from the try-block to another portion of code called the catch-block (along with the information in the value thrown.)**
- **Execution of the try-block <u>stops</u> when the throw statement is executed, and <u>execution</u> of the catch-block *that corresponds to type of the value thrown* is started. (There can be more than one catch block. Details shortly.)**
- **Executing the catch-block is known as catching the exception or handling the exception.**
- **In Display 16.2 the <u>catch-block</u> is:**

catch(int e)

{   cout<< e << " donuts, and No Milk!\n"<<"Go buy some milk.\n";   }

A Toy Example of Exception Handling (8 of 14)

- **The catch block looks and behaves very much like a function definition with a parameter of type int.**
- **Of course this is *not* a function definition, but there are similarities.**
- **The catch block is a separate piece of code that is executed in response to**

  throw some_int;
- **Instead of calling a function the response is to start execution of the catch block.**
- **The catch block is often referred to as an exception handler.**
- **The similarities to a function call are:**
  - **control flow is transferred to another piece of code**
  - **information is transferred to that piece of code.**

A Toy Example of Exception Handling (9 of 14)

- **Look at the catch block header:**

  catch(int e)
- **The identifier e looks and behaves like a function parameter.**
- **In fact we call the parameter e the catch-block parameter.**
- **The catch-block parameter does two things:**
  - **The catch-block parameter type specifies what type exception this catch-block can catch.**
  - **Upon starting the catch-block, the identifier e receives the value that is thrown by the throw statement.**
- **The catch-block parameter type enables choosing between catch-blocks corresponding to several exceptions which could be thrown. More in the text's section, "Multiple Throws and Catches".**
- **The identifier e gives a name in the catch-block to the value that was thrown and is caught.**
- **<u>Any legal C++ identifier may be used to name the catch-block parameter, even specific ExceptionType objects</u>.**

A Toy Example of Exception Handling (10 of 14)

- **Let's take a detailed look at the catch block from Display 16.2:**

  catch(int e)

  { cout << e << " donuts, and No Milk!\n" <<

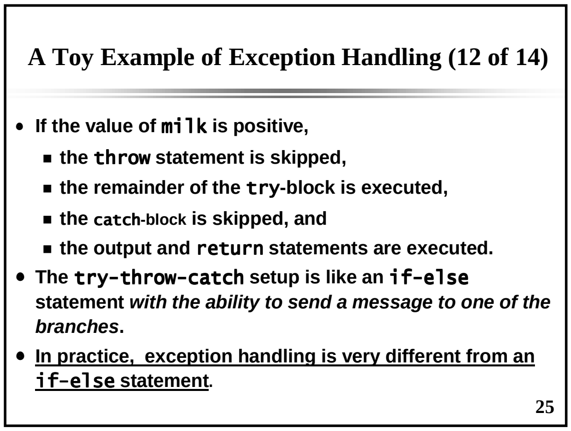
  "Go buy some milk.\n";

  }
- **When the exception is thrown, the type must be int for this catch- block to apply/get-executed.**
- **The throw statement sends the value of the variable donut which has type int.**
- **The catch-block parameter matches the type thrown, so the catch-block catches the value thrown.**

A Toy Example of Exception Handling (11 of 14)

- **Suppose that the value of 12 and the value of milk is 0:**
- **The value of milk is not positive, so the if statement executes the throw statement.**
- **When the catch-block is executed the value of donuts is plugged in for the catch-block parameter e, and this output is produced:**

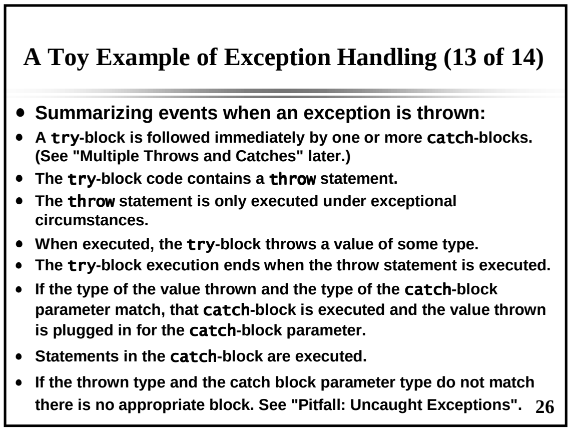
```
12 donuts, and No Milk!
Go buy some milk.
```



A Toy Example of Exception Handling (12 of 14)

- If the value of `milk` is positive,
  - the `throw` statement is skipped,
  - the remainder of the `try`-block is executed,
  - the `catch`-block is skipped, and
  - the output and `return` statements are executed.
- The `try-throw-catch` setup is like an `if-else` statement *with the ability to send a message to one of the branches*.
- In practice, exception handling is very different from an `if-else` statement.

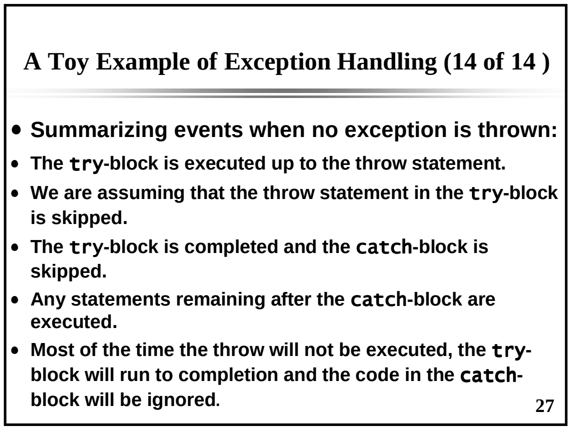
25



A Toy Example of Exception Handling (13 of 14)

- Summarizing events when an exception is thrown:
- A `try`-block is followed immediately by one or more `catch`-blocks. (See "Multiple Throws and Catches" later.)
- The `try`-block code contains a `throw` statement.
- The `throw` statement is only executed under exceptional circumstances.
- When executed, the `try`-block throws a value of some type.
- The `try`-block execution ends when the throw statement is executed.
- If the type of the value thrown and the type of the `catch`-block parameter match, that `catch`-block is executed and the value thrown is plugged in for the `catch`-block parameter.
- Statements in the `catch`-block are executed.
- If the thrown type and the catch block parameter type do not match there is no appropriate block. See "Pitfall: Uncaught Exceptions".

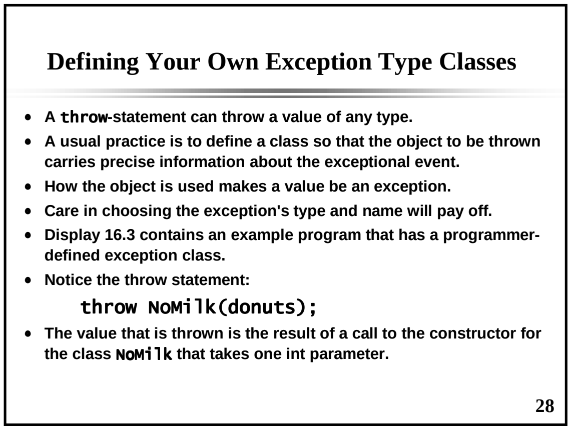
26



A Toy Example of Exception Handling (14 of 14)

- Summarizing events when no exception is thrown:
- The `try`-block is executed up to the throw statement.
- We are assuming that the throw statement in the `try`-block is skipped.
- The `try`-block is completed and the `catch`-block is skipped.
- Any statements remaining after the `catch`-block are executed.
- Most of the time the throw will not be executed, the `try`-block will run to completion and the code in the `catch`-block will be ignored.

27



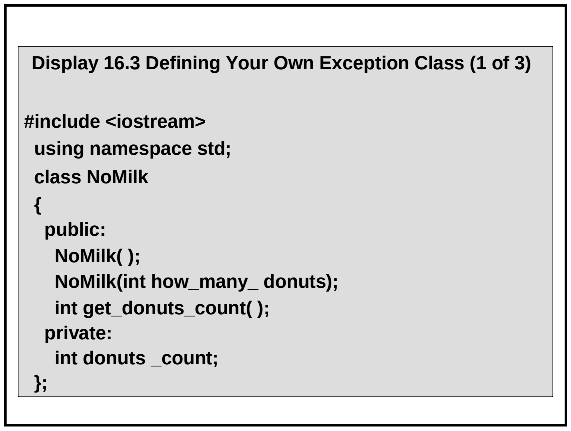
Defining Your Own Exception Type Classes

- A `throw`-statement can throw a value of any type.
- A usual practice is to define a class so that the object to be thrown carries precise information about the exceptional event.
- How the object is used makes a value be an exception.
- Care in choosing the exception's type and name will pay off.
- Display 16.3 contains an example program that has a programmer-defined exception class.
- Notice the throw statement:

```
throw NoMilk(donuts);
```

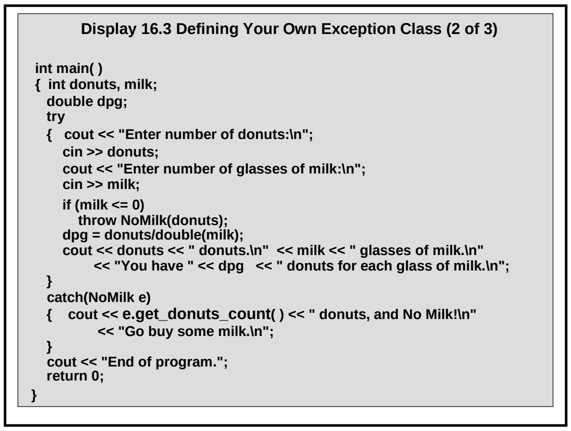
- The value that is thrown is the result of a call to the constructor for the class `NoMilk` that takes one int parameter.

28



Display 16.3 Defining Your Own Exception Class (1 of 3)

```
#include <iostream>
 using namespace std;
 class NoMilk
 {
   public:
     NoMilk( );
     NoMilk(int how_many_ donuts);
     int get_donuts_count( );
   private:
     int donuts _count;
 };
```



Display 16.3 Defining Your Own Exception Class (2 of 3)

```
int main( )
{  int donuts, milk;
   double dpg;
   try
   {   cout << "Enter number of donuts:\n";
       cin >> donuts;
       cout << "Enter number of glasses of milk:\n";
       cin >> milk;
       if (milk <= 0)
           throw NoMilk(donuts);
       dpg = donuts/double(milk);
       cout << donuts << " donuts.\n"  << milk << " glasses of milk.\n"
               << "You have " << dpg   << " donuts for each glass of milk.\n";
   }
   catch(NoMilk e)
   {    cout << e.get_donuts_count( ) << " donuts, and No Milk!\n"
                << "Go buy some milk.\n";
   }
   cout << "End of program.";
   return 0;
}
```

Display 16.3 Defining Your Own Exception Class (3 of 3)

```
NoMilk::NoMilk( )
{ }

NoMilk::NoMilk(int how_many) : count_donuts(how_many)
{ }

int NoMilk::get_donuts_count( )
{
   return donuts_count;
}
```

try-throw-catch

This is the basic mechanism for throwing and catching exceptions. The throw statement throws the exception (a value). The catch-block catches the exception (a value). When an exception is thrown, the try-block ends and then the code in the catch-block is executed. After the catch-block is complete, the code after the catch block(s) is executed, provided the catch-block has not ended the program or taken some other special action.

If no exception is thrown in the try-block then after the try-block is completed, program execution continues with the code after the catch-block(s). (In other words, then the catch-block(s) are ignored.)

Syntax:

```
try
{  Some_statements;
  <Either some code with a throw-statement or an invocation of
    a  function that might throw an exception.>
   Some_more_statements;
}
catch( Type e)
{   <Code to handle exception if a value of type Type is thrown.>  }
```

Multiple Throws and Catches

- A single `try`-block could throw any number of exception objects of several different types.
- In any one `try`-block only one exception will be thrown (because throwing an exception ends the `try`-block).
- Different types of exceptions may be thrown depending on events.
- While each `catch`-block can catch only one type of exception, the exact behavior can be tailored to the value of the exception.
- Display 16.4 has two `catch`-blocks for its one try-block.
- A coding note:

  In Display 16.4, there is no parameter for the `catch`-block for DivideByZero. The exception type communicates everything needed -- namely the fact that there is a divide by zero exception -- so there no need for a parameter, and we do not provide one.

Display 16.4 Catching Multiples Exceptions (1 of 3)

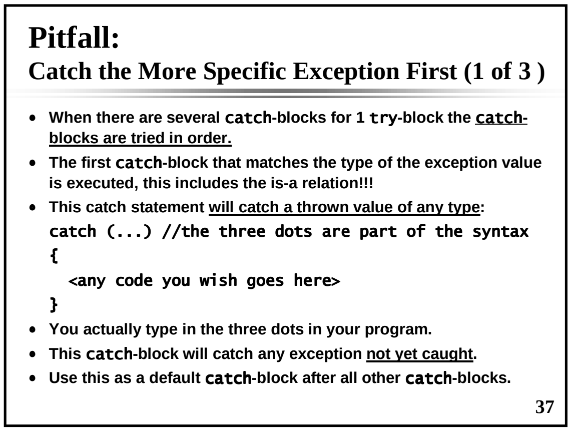
```
#include <iostream>
#include <string>
using namespace std;

class NegativeNumber
{
public:
   NegativeNumber( );
   NegativeNumber(string take_me_to_your_catch_block);
   string get_message( );
private:
   string message;
};

class DivideByZero
{         DivideByZero();      };
```

Display 16.4 Catching Multiples Exceptions (2 of 3)

```
int main( )
{   int jem_hadar, klingons;
   double portion;

   try
   {   cout << "Enter number of Jem Hadar warriors:\n";
       cin >> jem_hadar;
       if (jem_hadar < 0)
           throw NegativeNumber("Jem Hadar");
       cout << "How many Klingon warriors do you have?\n";
       cin >> klingons;
       if (klingons < 0)
          throw NegativeNumber("Klingons");
       if (klingons != 0)
          portion = jem_hadar/double(klingons);
       else
          throw DivideByZero( );
       cout << "Each Klingon must fight "   << portion << " Jem Hadar.\n";
   }
```

Display 16.4 Catching Multiples Exceptions (3 of 3)

```
 catch(NegativeNumber e)
 {  cout << "Cannot have a negative number of "
          << e.get_message( ) << endl;
  }
 catch(DivideByZero)
 {   cout << "C++ is fun and division by zero is forbidden.\n";   }
  cout << "End of program.\n";
  return 0;
}

NegativeNumber::NegativeNumber( )
{ }
 NegativeNumber::NegativeNumber(
          string take_me_to_your_catch_block)
                     : message(take_me_to_your_catch_block)
 { }
string NegativeNumber::get_message( )
{     return message;    }
```

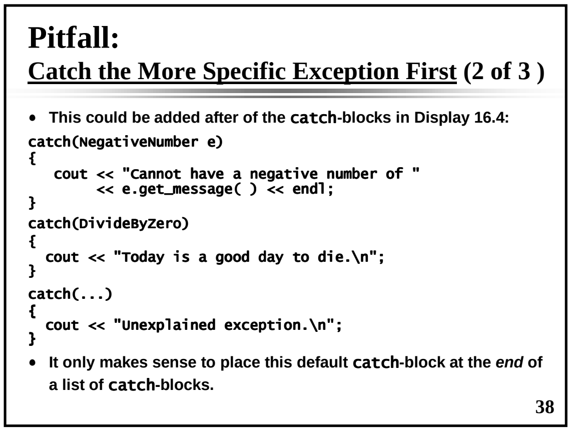


Pitfall:
Catch the More Specific Exception First (1 of 3)

- When there are several `catch`-blocks for 1 `try`-block the <u>`catch`-blocks are tried in order.</u>
- The first `catch`-block that matches the type of the exception value is executed, this includes the is-a relation!!!
- This catch statement <u>will catch a thrown value of any type</u>:

```
catch (...) //the three dots are part of the syntax
{
  <any code you wish goes here>
}
```

- You actually type in the three dots in your program.
- This `catch`-block will catch any exception <u>not yet caught</u>.
- Use this as a default `catch`-block after all other `catch`-blocks.

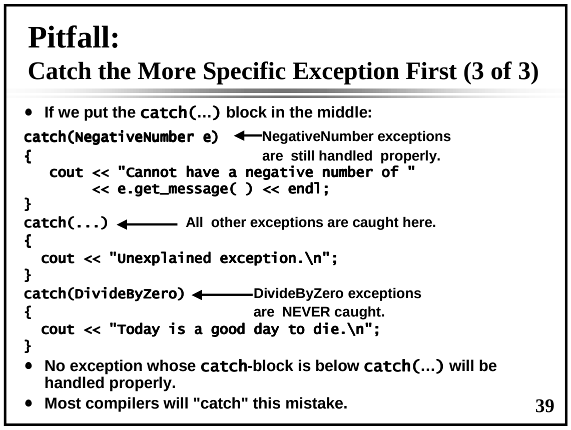


Pitfall:
<u>Catch the More Specific Exception First</u> (2 of 3)

- This could be added after of the `catch`-blocks in Display 16.4:

```
catch(NegativeNumber e)
{
   cout << "Cannot have a negative number of "
        << e.get_message( ) << endl;
}
catch(DivideByZero)
{
  cout << "Today is a good day to die.\n";
}
catch(...)
{
  cout << "Unexplained exception.\n";
}
```

- It only makes sense to place this default `catch`-block at the *end* of a list of `catch`-blocks.

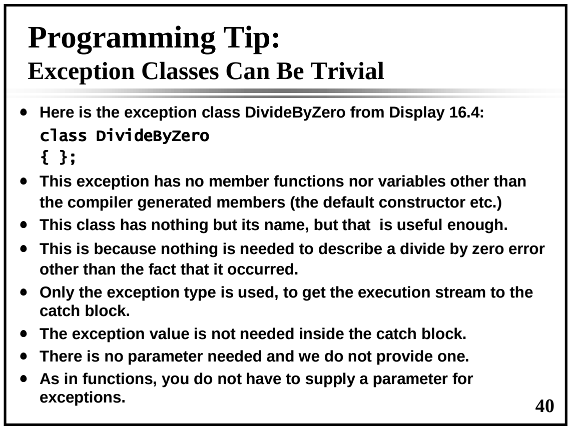


Pitfall:
Catch the More Specific Exception First (3 of 3)

- If we put the `catch(...)` block in the middle:

```
catch(NegativeNumber e)  ←NegativeNumber exceptions
{                          are  still handled  properly.
   cout << "Cannot have a negative number of "
        << e.get_message( ) << endl;
}
catch(...)  ←———  All  other exceptions are caught here.
{
  cout << "Unexplained exception.\n";
}
catch(DivideByZero) ←———DivideByZero exceptions
{                        are  NEVER caught.
  cout << "Today is a good day to die.\n";
}
```

- No exception whose `catch`-block is below `catch(...)` will be handled properly.
- Most compilers will "catch" this mistake.

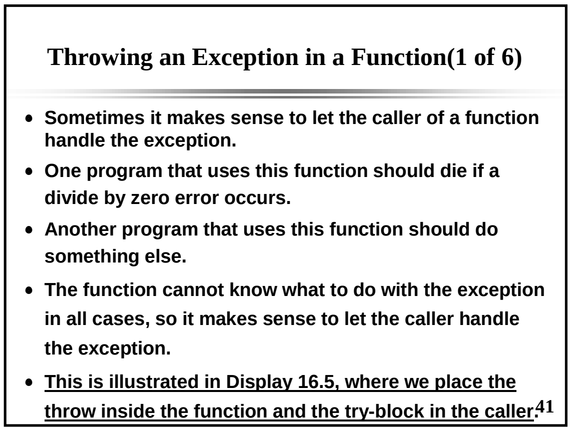


Programming Tip:
Exception Classes Can Be Trivial

- Here is the exception class DivideByZero from Display 16.4:

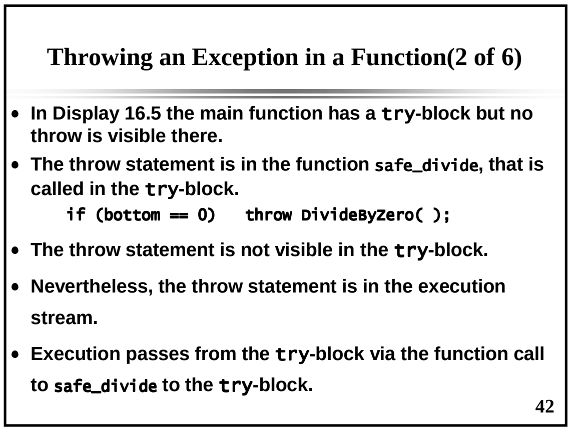
```
class DivideByZero
{ };
```

- This exception has no member functions nor variables other than the compiler generated members (the default constructor etc.)
- This class has nothing but its name, but that is useful enough.
- This is because nothing is needed to describe a divide by zero error other than the fact that it occurred.
- Only the exception type is used, to get the execution stream to the catch block.
- The exception value is not needed inside the catch block.
- There is no parameter needed and we do not provide one.
- As in functions, you do not have to supply a parameter for exceptions.



Throwing an Exception in a Function(1 of 6)

- Sometimes it makes sense to let the caller of a function handle the exception.
- One program that uses this function should die if a divide by zero error occurs.
- Another program that uses this function should do something else.
- The function cannot know what to do with the exception in all cases, so it makes sense to let the caller handle the exception.
- <u>This is illustrated in Display 16.5, where we place the throw inside the function and the try-block in the caller.</u>

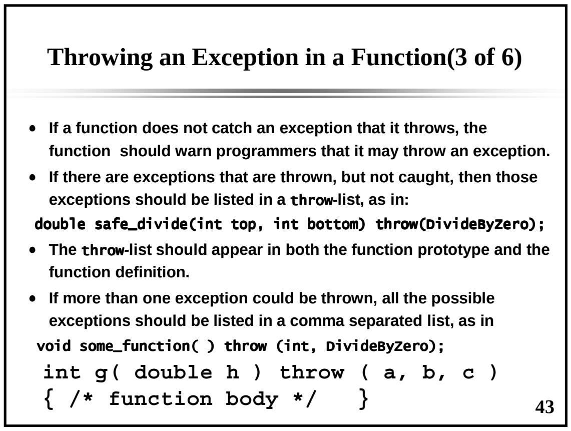


Throwing an Exception in a Function(2 of 6)

- In Display 16.5 the main function has a `try`-block but no throw is visible there.
- The throw statement is in the function `safe_divide`, that is called in the `try`-block.

```
    if (bottom == 0)   throw DivideByZero( );
```

- The throw statement is not visible in the `try`-block.
- Nevertheless, the throw statement is in the execution stream.
- Execution passes from the `try`-block via the function call to `safe_divide` to the `try`-block.



Throwing an Exception in a Function(3 of 6)

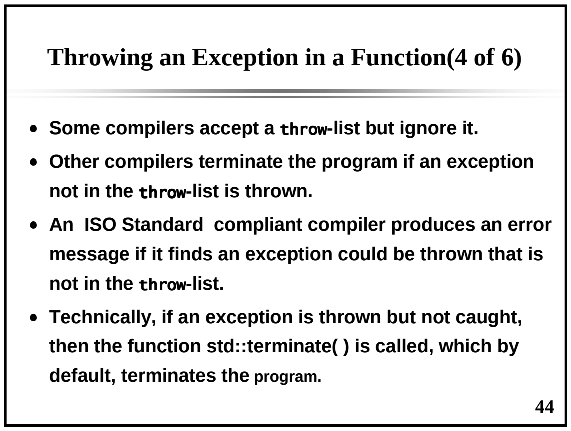
- If a function does not catch an exception that it throws, the function should warn programmers that it may throw an exception.
- If there are exceptions that are thrown, but not caught, then those exceptions should be listed in a `throw`-list, as in:

```
double safe_divide(int top, int bottom) throw(DivideByZero);
```

- The `throw`-list should appear in both the function prototype and the function definition.
- If more than one exception could be thrown, all the possible exceptions should be listed in a comma separated list, as in

```
void some_function( ) throw (int, DivideByZero);
 int g( double h ) throw ( a, b, c )
 { /* function body */   }
```

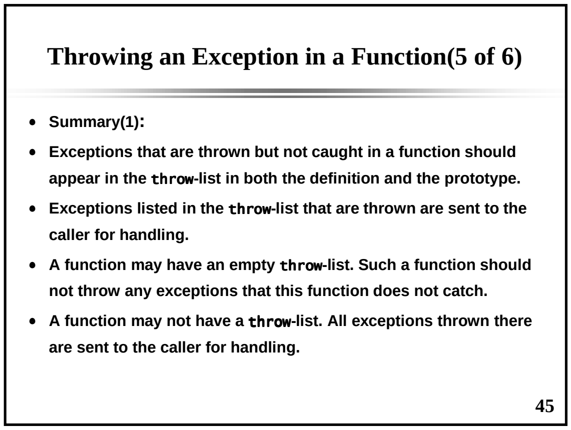
43



Throwing an Exception in a Function(4 of 6)

- Some compilers accept a `throw`-list but ignore it.
- Other compilers terminate the program if an exception not in the `throw`-list is thrown.
- An ISO Standard compliant compiler produces an error message if it finds an exception could be thrown that is not in the `throw`-list.
- Technically, if an exception is thrown but not caught, then the function std::terminate( ) is called, which by default, terminates the program.

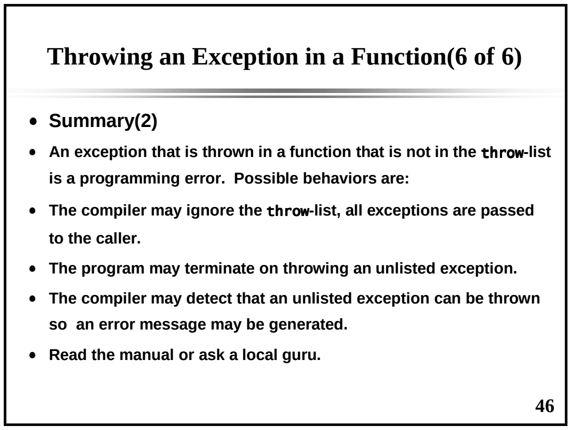
44



Throwing an Exception in a Function(5 of 6)

- Summary(1):
- Exceptions that are thrown but not caught in a function should appear in the `throw`-list in both the definition and the prototype.
- Exceptions listed in the `throw`-list that are thrown are sent to the caller for handling.
- A function may have an empty `throw`-list. Such a function should not throw any exceptions that this function does not catch.
- A function may not have a `throw`-list. All exceptions thrown there are sent to the caller for handling.

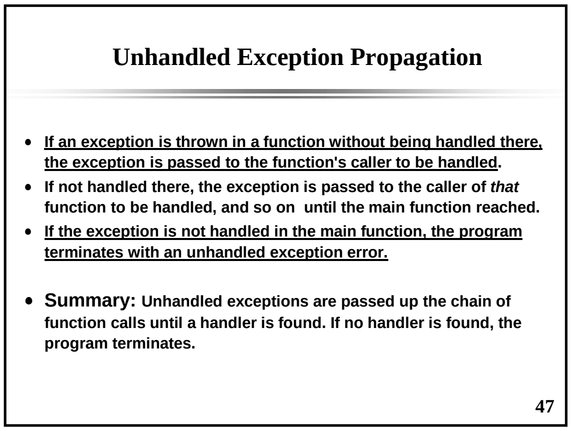
45



Throwing an Exception in a Function(6 of 6)

- Summary(2)
- An exception that is thrown in a function that is not in the `throw`-list is a programming error. Possible behaviors are:
- The compiler may ignore the `throw`-list, all exceptions are passed to the caller.
- The program may terminate on throwing an unlisted exception.
- The compiler may detect that an unlisted exception can be thrown so an error message may be generated.
- Read the manual or ask a local guru.

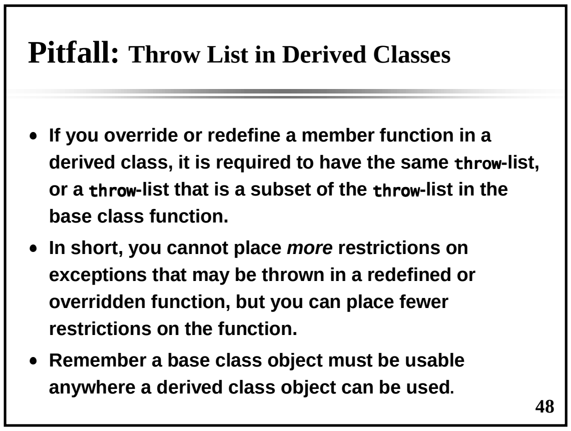
46



Unhandled Exception Propagation

- If an exception is thrown in a function without being handled there, the exception is passed to the function's caller to be handled.
- If not handled there, the exception is passed to the caller of *that* function to be handled, and so on until the main function reached.
- If the exception is not handled in the main function, the program terminates with an unhandled exception error.
- Summary: Unhandled exceptions are passed up the chain of function calls until a handler is found. If no handler is found, the program terminates.

47



Pitfall: Throw List in Derived Classes

- If you override or redefine a member function in a derived class, it is required to have the same `throw`-list, or a `throw`-list that is a subset of the `throw`-list in the base class function.
- In short, you cannot place *more* restrictions on exceptions that may be thrown in a redefined or overridden function, but you can place fewer restrictions on the function.
- Remember a base class object must be usable anywhere a derived class object can be used.

48

16.2 Programming Techniques for Exception Handling

- **We have explained HOW exception handling works.**
- **We have NOT given you any examples of how to make realistic use of exception handling.**
- **When do you throw exceptions?**

49

When to Throw an Exception (1 of 3)

- **Two cases arise:**

(1) You have a function where you want to throw an exception. There you should have a throw-list that lists all the exceptions that may be thrown.

```
void  func_A( ) throw (MyException)
{  ...
    throw MyException(<argument_if_needed>);
    ...
 }
```

50

When to Throw an Exception (2 of 3)

(2) You have a function that calls some other function that throws an exception you want to catch:

```
void  funcB( )
{
  ...
  try
  {  throw MyException(<argument_if_needed>);
     ...
   }
  catch( MyException e)
  {
     <Handle_exception>
   }
  ...
}
```

51

When to Throw an Exception (3 of 3)

- **If a problem can be handled in some other way, do not use exceptions.**
- **Reserve exceptions for cases where use of exceptions is unavoidable.**

When to throw an Exception

For the most part, `throw`-statements should be used within functions and listed in the `throw`-list for the function. Moreover, they should be reserved for situations in which the way the exceptional situation is handled depends on how and where the function is used. If the way that the exceptional condition is handled depends on how and where the function is invoked, then the best thing to do is to let the programmer who invokes the function handle the exception. In all other situations it is almost always preferable to avoid throwing an exception.

52

Pitfall: Uncaught Exceptions

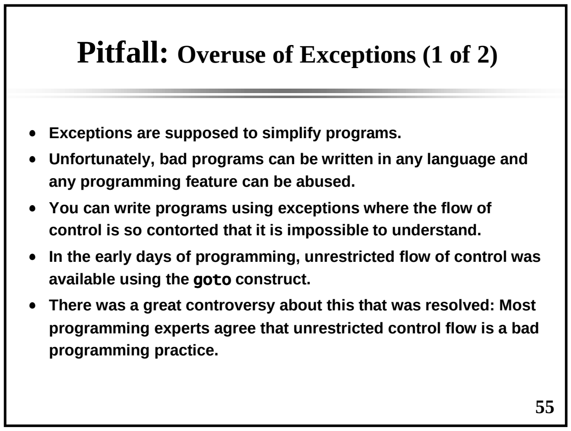
- **Every exception thrown by your program should be caught some place in your program.**
- **The (default) penalty for not catching an exception that is thrown is termination of your program.**
- **The std::terminate( ) function is called by default, but you can change the default behavior. How to do this is beyond the scope of the text.**

53

Pitfall: Nested `Try-Catch` Blocks

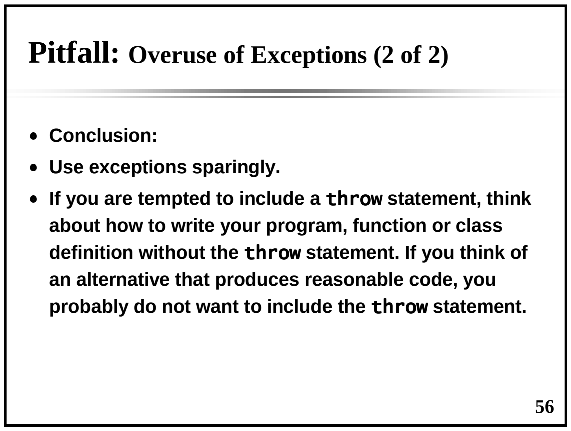
- **You *can* nest `try-throw-catch` sequence inside another `try`-block, or inside a `catch`-block for that matter.**
- **This *may rarely* be useful but if you are tempted to do this, look for a nicer way to organize your program.**
- **It is almost always better to place the inner try-catch sequence inside a function and call that from the try or catch block where you are tempted to nest it. (It may be better to just eliminate the inner try and move the catch to someplace in the catch blocks below.)**
- **An exception thrown in an inner try block and not caught in the catch block belonging to the inner try block is passed to the outer try block and may be caught by one of its catch blocks.**

54



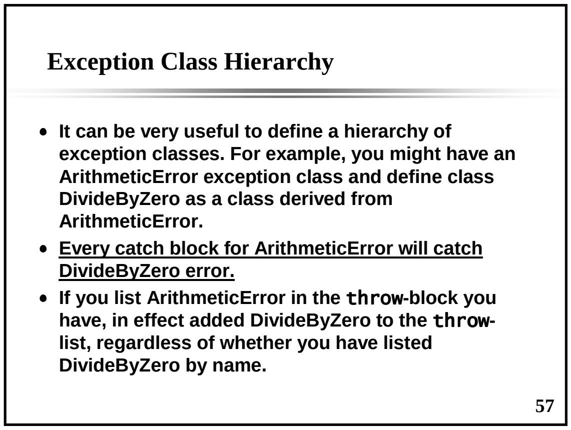
Pitfall: Overuse of Exceptions (1 of 2)

- Exceptions are supposed to simplify programs.
- Unfortunately, bad programs can be written in any language and any programming feature can be abused.
- You can write programs using exceptions where the flow of control is so contorted that it is impossible to understand.
- In the early days of programming, unrestricted flow of control was available using the `goto` construct.
- There was a great controversy about this that was resolved: Most programming experts agree that unrestricted control flow is a bad programming practice.



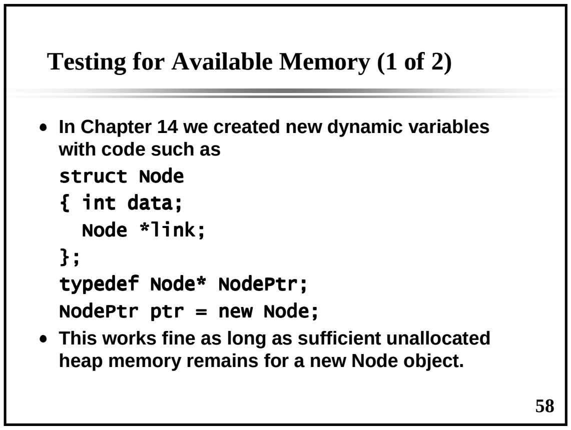
Pitfall: Overuse of Exceptions (2 of 2)

- Conclusion:
- Use exceptions sparingly.
- If you are tempted to include a `throw` statement, think about how to write your program, function or class definition without the `throw` statement. If you think of an alternative that produces reasonable code, you probably do not want to include the `throw` statement.



Exception Class Hierarchy

- It can be very useful to define a hierarchy of exception classes. For example, you might have an ArithmeticError exception class and define class DivideByZero as a class derived from ArithmeticError.
- Every catch block for ArithmeticError will catch DivideByZero error.
- If you list ArithmeticError in the `throw`-block you have, in effect added DivideByZero to the `throw`-list, regardless of whether you have listed DivideByZero by name.

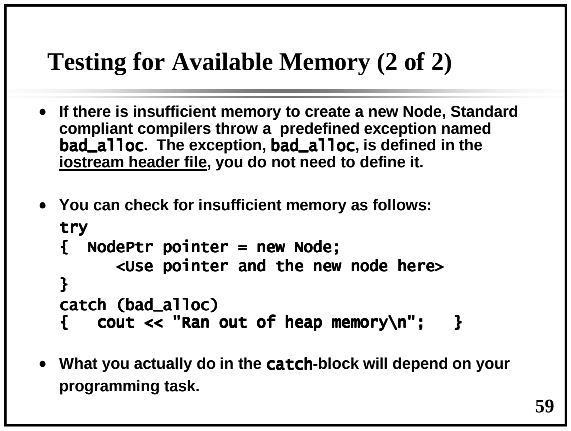


Testing for Available Memory (1 of 2)

- In Chapter 14 we created new dynamic variables with code such as

```
struct Node
{ int data;
  Node *link;
};
typedef Node* NodePtr;
NodePtr ptr = new Node;
```

- This works fine as long as sufficient unallocated heap memory remains for a new Node object.



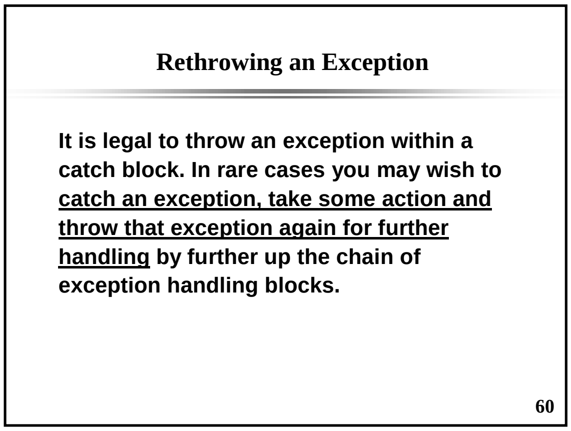
Testing for Available Memory (2 of 2)

- If there is insufficient memory to create a new Node, Standard compliant compilers throw a predefined exception named `bad_alloc`. The exception, `bad_alloc`, is defined in the iostream header file, you do not need to define it.

- You can check for insufficient memory as follows:

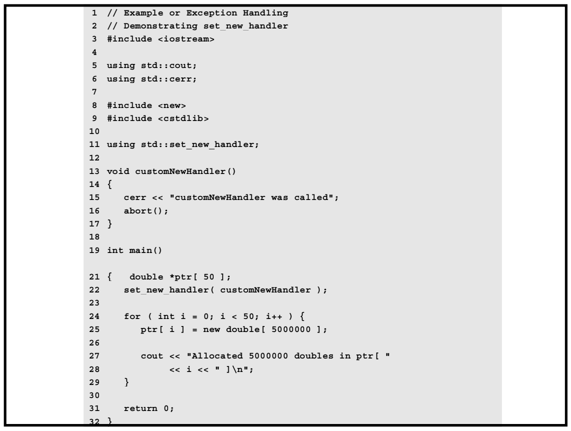
```
try
{  NodePtr pointer = new Node;
      <Use pointer and the new node here>
}
catch (bad_alloc)
{   cout << "Ran out of heap memory\n";   }
```

- What you actually do in the `catch`-block will depend on your programming task.

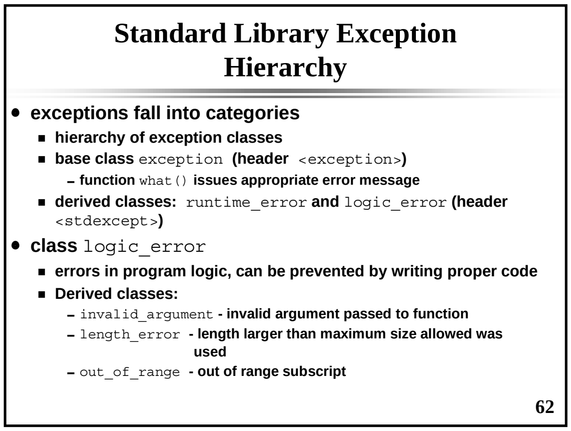


Rethrowing an Exception

It is legal to throw an exception within a catch block. In rare cases you may wish to catch an exception, take some action and throw that exception again for further handling by further up the chain of exception handling blocks.

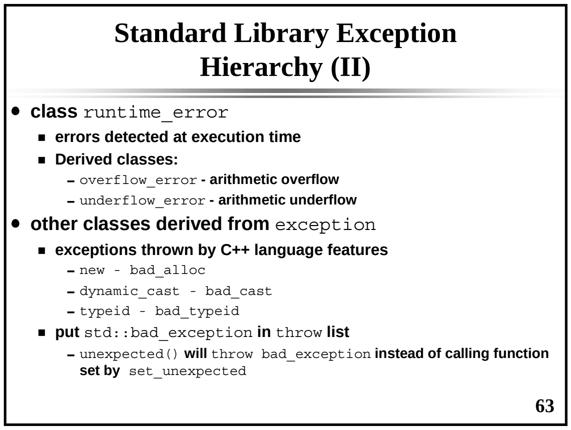


```
// Example or Exception Handling
// Demonstrating set_new_handler
#include <iostream>

using std::cout;
using std::cerr;

#include <new>
#include <cstdlib>

using std::set_new_handler;

void customNewHandler()
{
   cerr << "customNewHandler was called";
   abort();
}

int main()

{   double *ptr[ 50 ];
   set_new_handler( customNewHandler );

   for ( int i = 0; i < 50; i++ ) {
      ptr[ i ] = new double[ 5000000 ];

      cout << "Allocated 5000000 doubles in ptr[ "
           << i << " ]\n";
   }

   return 0;
}
```



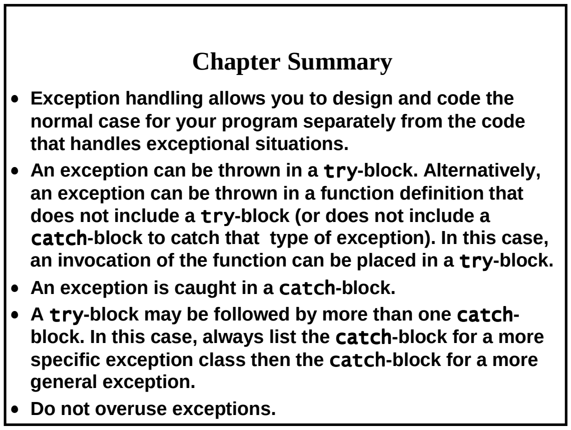
Standard Library Exception Hierarchy

- **exceptions fall into categories**
  - **hierarchy of exception classes**
  - **base class** `exception` **(header** `<exception>`**)**
    - **function** `what()` **issues appropriate error message**
  - **derived classes:** `runtime_error` **and** `logic_error` **(header** `<stdexcept>`**)**
- **class** `logic_error`
  - **errors in program logic, can be prevented by writing proper code**
  - **Derived classes:**
    - `invalid_argument` **- invalid argument passed to function**
    - `length_error` **- length larger than maximum size allowed was used**
    - `out_of_range` **- out of range subscript**



Standard Library Exception Hierarchy (II)

- **class** `runtime_error`
  - **errors detected at execution time**
  - **Derived classes:**
    - `overflow_error` **- arithmetic overflow**
    - `underflow_error` **- arithmetic underflow**
- **other classes derived from** `exception`
  - **exceptions thrown by C++ language features**
    - `new - bad_alloc`
    - `dynamic_cast - bad_cast`
    - `typeid - bad_typeid`
  - **put** `std::bad_exception` **in** `throw` **list**
    - `unexpected()` **will** `throw bad_exception` **instead of calling function set by** `set_unexpected`



Chapter Summary

- **Exception handling allows you to design and code the normal case for your program separately from the code that handles exceptional situations.**
- **An exception can be thrown in a `try`-block. Alternatively, an exception can be thrown in a function definition that does not include a `try`-block (or does not include a `catch`-block to catch that type of exception). In this case, an invocation of the function can be placed in a `try`-block.**
- **An exception is caught in a `catch`-block.**
- **A `try`-block may be followed by more than one `catch`-block. In this case, always list the `catch`-block for a more specific exception class then the `catch`-block for a more general exception.**
- **Do not overuse exceptions.**