**EXCEPTION HANDLING**

Can resolve exceptions

* Allow a program to continue executing or
* Notify the user of the problem and
* Terminate the program in a controlled manner

Makes programs robust and fault-tolerant.

open file

All I want is green ones but I have to handle red ones too. This makes your program difficult to read, modify, maintain and debug.

check if file is open

allocate memory

check if the allocate is success

open a network connection

connected?

You can say:

try{

If anything happens in try block, an exception is thrown automatically and that exception will be handled in the catch part.

Now you are not running ifs anymore so this code is more efficient.

open file

allocate memory

open a network connection

}

catch(exception){

file error

memory error

network error

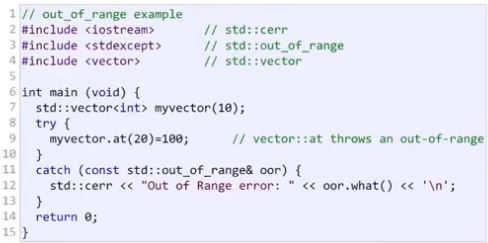
}

Exception handling:

* Removes error-handling code from the program execution’s “main line”
* Programmers can handle any exceptions they choose
  + All exceptions
  + All exceptions of a certain type
  + All exceptions of a group of related types

Class **exception**

* Is the standard C++ base class for all exceptions
* Provides its derived classes with **virtual** function **what**
  + Returns the exception’s stored error message



**CHECK DivideByZeroException.h, driver.cpp in order.**

If an exception occurs and you don’t handle it, then your program will be stopped and you’ll get the unhandled exception runtime error. This means that some of your code cause an exception and nobody handled it. So it is terminating your program. bc there is an exception that wasn’t handled properly.

try blocks should enclose:

* Statements that might cause exceptions
* Statements that should be skipped in case of an exception

Let’s say you call “sort()” function in try block and that sort function use “at()” vector function. If you don’t handle the out\_of\_range exception in sort function, it throws out\_of\_range exception and you have to handle it. So if you don’t handle it, whoever calls you should handle it. If nobody handles it, your program will be terminated.

*So all thrown exceptions should be handled.*

catch handlers

* One or more catch handlers for each try block
  + You can have more than one exceptions with several catch blocks
  + You cannot catch same type in 2 different catch handlers following a single try block
  + Each catch handler can have only a single parameter
* Parameter represents the type of exception to process
* Executes if exception parameter type matches the exception thrown in the try block
  + Could be a base class of the thrown exception’s class

Don’t write your regular part of your algorithm depending on the exceptions. Exceptions mean that sth bad happened and I cannot continue.

Avoid using exception handling as an alternate form of flow of control. These “additional” exceptions can “get in the way” of genuine error-type exceptions. Don’t do useful stuffs with the exceptions. Only handle exceptions.

After an expection is handled, control will not return to the first statement after the throw point. Try block will be terminated if it throws sth.

With exception handling, a program can continue executing (rather than terminating) after dealing with a problem. This helps ensure the kind of robust applications that contribute to what is called mission-critical computing or business-critical computing.

Associating each type of runtime error with an appropriately named exception object improves program clarity.

In fact you don’t have to make up a new exception for all the errors bc most of the time there is always standard exception defined for it.

Don’t leave error handling to later stages. Always develop your program with the exception handling in mind.

throw operand can be of any type

* If the throw operand is an object, it is called an exception object
* Don’t throw int, string, etc. anyway.
* The throw operand initializes the exception parameter in the matching catch handler, if one is found

**Rethrowing an exception**

An exception happened, I am catching the exception, I close my files, I deallocate my memory, I close my network connection etc., and I rethrow that exception bc I don’t know how to handle it.

– Empty “throw;” statement

– Use when a “catch” handler cannot or can only partially process an exception

– Next enclosing “try” block attempts to match the exception with one of its “catch” handlers

try{

…

}

catch(...){

cout << “Something bad happened.”;

throw;

//Probably nobody is gonna catch this exception and handle it.

//My program will be terminated but not in a bad way.

//It is gonna be terminated with a message saying that:

// Something bad happened.

}

catch(…) 🡪 means that catch everything if some exception is thrown.

Executing an empty throw statement that is situated outside a catch handler causes a call to function terminate, which abandons exception processing and terminates the program immediately.

**CHECK throw.cpp.**

If a function doesn’t catch the thrown exception, whoever calls it has to catch that exception. This goes all the way up to main function. If main function doesn’t catch it, then function terminate will be called and it is going to call unhandled exception.

**Exception Specifications (a.k.a. throw lists)**

int someFunction(double value)

throw(ExceptionA, ExceptionB, ExceptionC)

{

…

}

Indicates someFunction can throw exceptions of types ExceptionA, ExceptionB, and ExceptionC.

Deprecated with C++11 !!!

But you can do this:

int someFunction(double value) noexcept

{

…

}

This means that someFunction will not throw any exception. So you don’t need to write a catch statement for this function.

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.

Throwing an exception that has not been declared in a function’s exception specification causes a call to function unexpected.

Not very important

* Calls the function registered with function set\_unexpected
* Function terminate is called by default

Function set\_unexpected of <exception>

* Takes as argument a pointer to a function with no arguments and a void return type.
* Returns a pointer to the last function called by unexpected
  + Returns 0 the first time

Function terminate

* Called when:
  + No matching catch is found for a thrown exception
  + A destructor attempts to throw an exception during stack unwinding
  + Attempting to rethrow an exception when no exception is being handled
  + Calling function unexpected before registering a function with function set\_unexpected
* Calls the function registered with function set\_terminate
* Function abort is called by default

Function set\_terminate

* Takes as argument a pointer to a function with no arguments and a void return type.
* Returns a pointer to the last function called by terminate
* Returns 0 the first time

Function abort

* Terminates the program without calling destructors for automatic or static storage class objects
  + Could lead to resource leaks

int main(){

quotient(1, 0);

return 0;

}

quotient will throw an exception but nobody handles it. There is a global function named unexpected. It is going to be called. Job of that unexpected is terminating the program and printing the message saying that “This program causes an unhandled exception and I am terminating the program. Here is your stack.”

**STACK UNWINDING**

* Occurs when a thrown exception is not caught in a particular scope
* Unwinding a function terminates that function
  + All local variables of the function are destroyed
  + Control returns to the statement that invoked the function
* All the objects created on the stack will be terminated properly
* All the allocated memory for the constructor will be deallocated.
* Attempts are made to catch the exception in outer try…catch blocks
* If the exception is never caught, function terminate is called

**CHECK unwinding.cpp.**

**EXCEPTIONS AND CONSTRUCTORS**

* Exceptions enable constructors, which cannot return values, to report errors to the program
* Exceptions thrown by constructors cause any already-constructed component objects to call their destructors
  + Only those objects that have already been constructed will be destructed

**EXCEPTIONS AND DESTRUCTORS**

* Destructors are called for all automatic objects in the terminated try block when an exception is thrown
  + Acquired resources can be placed in local objects to automatically release the resources when an exception occurs
* If a destructor invoked by stack unwinding throws an exception, function terminate is called

When an exception is thrown from the constructor for an object that is created in a new expression, the dynamically allocated memory for that object is released.

**Inheritance with exception classes:**

* New exception classes can be defined to inherit from existing exception classes
* A catch handler for a particular exception class can also catch exceptions of classes derived from that class

**new failures**

* Some compilers throw a bad\_alloc exception
  + Compliant to the C++ standard specification
* Some compilers return 0
  + C++ standard-compliant compilers also have a version of new that returns 0
    - Use expression new( nothrow ), where nothrow is of type nothrow\_t
* Some compilers throw bad\_alloc if <new> is included

**CHECK newReturn0.cpp, newThrowBad.cpp in order.**

**STANDARD LIBRARY EXCEPTION HIERARCHY**

Exception hierarchy classes

* Base-class **exception**
  + Contains virtual function what for storing error messages
  + Exception classes derived from exception:
    - **bad\_alloc** 🡪 thrown by new
    - **bad\_cast** 🡪 thrown by dynamic\_cast

Pet& pr = Dog();

Dog& dr = dynamic\_cast<Dog &> pr;

-------------------------------------------------

* + - * If I try to type cast base class reference to derived class reference, I use dynamic cast. References either return a reference or nothing else. You cannot return nullreference. There is no such thing. So either you return a Dog reference or throw an exception. That is bad\_cast exception.
    - **bad\_typeid** 🡪 thrown by typeid
    - **bad\_exception** 🡪 thrown by unexpected
      * Instead of terminating the program or calling the function specified by set\_unexpected
      * Used only if bad\_exception is in the function’s throw list

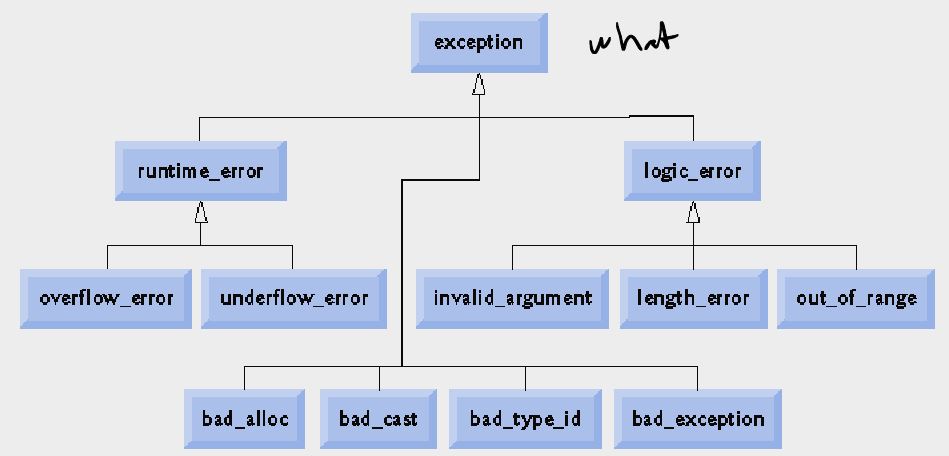
Class logic\_error, derived from exception

* Indicates errors in program logic
* Exception classes derived from logic\_error:
  + **invalid\_argument**
    - Indicates an invalid argument to a function
  + **length\_error**
    - Indicates a length larger than the maximum size for some object was used
  + **out\_of\_range**
    - Indicates a value, such as an array subscript, exceeded its allowed range

Class runtime\_error, derived from exception

* Indicates execution-time errors
* Exception classes derived from runtime\_error
  + **overflow\_error**
    - Indicates an arithmetic overflow error - an arithmetic result is larger than the largest storable number.
  + **underflow\_error**
    - Indicates an arithmetic underflow error - an arithmetic result is smaller than the smallest storable number

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 base-class catch catches all objects of classes derived from that base class, so the derived-class catch will never execute.



Usually if you throw an exception, choose one of these.

You can catch all of the exceptions by catching base class exception.

Programmer-defined exception classes need not be derived from class exception. Thus, writing **catch( exception anyException )** 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.

try{

…

}

catch(memory error){

…

}

catch(bad\_typeid){

…

}

catch(...){

cerr << “Send email to developer.”;

}

Put your most general error at the bottom.

The standard exception hierarchy is a good starting point for creating exceptions. Programmers can build programs that can throw standard exceptions, throw exceptions derived from the standard exceptions or throw their own exceptions not derived from the standard exceptions.

Use catch(...) to perform recovery that does not depend on the exception type (e.g., releasing common resources). The exception can be rethrown to alert more specific enclosing catch handlers.