**Exception Handling in C++**

# Sources of errors

# Compile-time errors

## Syntax errors

A syntax error is an error in the syntax of a sequence of characters or tokens that is intended to be written in a particular programming language.

For compiled languages, syntax errors are detected at compile-time. A program will not compile until all syntax errors are corrected.

For interpreted languages, however, a syntax error may be detected during program execution, and an interpreter's error messages might not differentiate syntax errors from errors of other kinds.

## Type errors

## Non-errors

# Link-time errors

# Run-time errors

## The caller deals with errors

## The callee deals with errors

## Error reporting

# Logic errors

A logic error produces unintended or undesired output or other behavior, but not to terminate abnormally (or crash), although it may not immediately be recognized as such.

Logic errors occur in both compiled and interpreted languages. Unlike a program with a syntax error, a program with a logic error is a valid program in the language, though it does not behave as intended. Often the only clue to the existence of logic errors is the production of wrong solutions, though static analysis may sometimes spot them.

Examples

Function in C to calculate the average of two numbers contains a logic error. It is missing parentheses in the calculation, so it compiles and runs but does not give the expected answer due to operator precedence (division is evaluated before addition).

float average(float a, float b) {

return a + b / 2; /\* should be (a + b) / 2 \*/

}

# Exceptions

Exceptions are runtime anomalies or abnormal conditions that a program encounters during its execution.

Exception handling provides a mechanism to decouple handling of errors or other exceptional circumstances from the typical control flow of your code.

## Bad arguments

## Range errors

## Bad input

## Narrowing errors

# Why Exception Handling?

Advantages of exception handling over traditional error handling.

## Separation of Error Handling code from Normal Code

In traditional error handling codes, code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable.

With try catch blocks, the code for error handling becomes separate from the normal flow.

## Functions/Methods can handle any exceptions they choose

A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller.

In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it)

## Grouping of Error Types

In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

# Exception Handling in C++

1. try block is used to execute the code and catch block is used to catch an exception.
2. 'catch all' catch (…) can be used to catch all types of exceptions.
3. Implicit type conversion doesn’t happen for primitive types.
4. If an exception is thrown and not caught anywhere, the program terminates abnormally.
5. **A derived class exception should be caught before a base class exception.**
6. Like Java, C++ library has a standard exception class which is base class for all standard exceptions. All objects thrown by components of the standard library are derived from this class. Therefore, all standard exceptions can be caught by catching this type.
7. In C++, try-catch blocks can be nested. Also, an exception can be re-thrown using "throw;".

Any further catch clauses for the same try block are still ignored – the throw causes the exception to go to the exception handlers in the next-higher context.

1. When an exception is thrown, all objects successfully created inside the enclosing try block are destructed before the control is transferred to catch block.

**Note:**

* **It is compiler error to put catch all block before any other catch. The catch(...) must be the last catch block.**
* There is a standard exception class like Exception class in Java.
* **All exceptions are unchecked in C++, i.e., compiler doesn't check if the exceptions are caught or not.**
* In C++, a function can specify the list of exceptions that it can throw using comma separated list like following.
  + void fun(int a, char b) **throw (Exception1, Exception2)**
  + void fun(int a, char b) **throw () // no exception thrown**
  + void fun(int a, char b) **// any type of exception may be thrown**
* C++ compiler doesn't check enforce a function to list the exceptions that it can throw. In Java, it is enforced. It is up to the programmer to specify. A programmer should specify the list.

**What should be put in a try block?**

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

#include <iostream>

using namespace std;

int main() {

int x = -1;

cout << "Before try" << endl;

try {

cout << "Inside try" << endl;

if (x < 0) {

throw x;

cout << "After throw (Never executed)" << endl;

}

}

catch (int x ) {

cout << "Exception Caught" << endl;

}

cout << "After catch (Will be executed)" << endl;

return 0;

}

Output:

Before try

Inside try

Exception Caught

After catch (Will be executed)

# Advantage

1. Remove error-handling code from the software's main line of code.
2. A method writer can choose to handle certain exceptions and delegate others to the caller.
3. An exception that occurs in a function can be handled anywhere in the function call stack.

# Catching base and derived classes as exceptions

If both base and derived classes are caught as exceptions then catch block of derived class must appear before the base class.

If we put base class first, then the derived class catch block will never be reached.

In C++, compiler might give warning about it, but compiles the code.

In Java, catching a base class exception before derived is not allowed by the compiler itself.

// catching base class exception before child class

#include<iostream>

using namespace std;

class Base {};

class Derived: public Base {};

int main() {

Derived d;

// some other stuff

try {

// Some monitored code

throw d;

}

catch(Base b) {

cout<<"Caught Base Exception";

}

catch(Derived d) { //This catch block is NEVER executed

cout<<"Caught Derived Exception";

}

return 0;

}

Output:

Caught Base Exception

// CW

Warnings:

prog.cpp: In function 'int main()':

prog.cpp:17:2: warning: exception of type 'Derived' will be caught

catch(Derived d) { //This catch block is NEVER executed

^

prog.cpp:14:2: warning: by earlier handler for 'Base'

catch(Base b) {

^

// catching base class exception after child class

#include<iostream>

using namespace std;

class Base {};

class Derived: public Base {};

int main() {

Derived d;

// some other stuff

try {

// Some monitored code

throw d;

}

catch(Derived d) {

cout<<"Caught Derived Exception";

}

catch(Base b) {

cout<<"Caught Base Exception";

}

return 0;

}

Output:

Caught Derived Exception

# Catch block and type conversion in C++

Implicit type conversion doesn’t happen for primitive types.

The derived type objects are converted to base type when a derived object is thrown and there is a catch block to catch base type.

#include <iostream>

using namespace std;

int main() {

try {

throw 'x';

}

catch(int x) {

cout << " Caught int " << x;

}

catch(...) {

cout << "Default catch block";

}

return 0;

}

Output:

Default catch block

In the above program, a character ‘x’ is thrown and there is a catch block to catch an int. One might think that the int catch block could be matched by considering ASCII value of ‘x’. But such conversions are not performed for catch blocks.

#include <iostream>

using namespace std;

class MyExcept1 {};

class MyExcept2 {

public:

// Conversion constructor

MyExcept2 (const MyExcept1 &e ) {

cout << "Conversion constructor called";

}

};

int main() {

try {

MyExcept1 myexp1;

throw myexp1;

}

catch(MyExcept2 e2) {

cout << "Caught MyExcept2 " << endl;

}

catch(...) {

cout << " Default catch block " << endl;

}

return 0;

}

Output:

Default catch block

# Exception handling and object destruction

When an exception is thrown, destructor of the objects (whose scope ends with the try block) is automatically called before the catch block gets executed.

Destructor is only called for the completely constructed objects. When constructor of an object throws an exception, destructor for that object is not called.

#include <iostream>

using namespace std;

class Test {

public:

Test() { cout << "Constructing an object of Test " << endl; }

~Test() { cout << "Destructing an object of Test " << endl; }

};

int main() {

try {

Test t1;

throw 10;

}

catch(int i) {

cout << "Caught " << i << endl;

}

return 0;

}

Output:

Constructing an object of Test

Destructing an object of Test

Caught 10

#include <iostream>

using namespace std;

class Test1 {

public:

Test1() { cout << "Constructing an Object of Test1" << endl; }

~Test1() { cout << "Destructing an Object of Test1" << endl; }

};

class Test2 {

public:

// Following constructor throws an integer exception

Test2() { cout << "Constructing an Object of Test2" << endl; throw 20; }

~Test2() { cout << "Destructing an Object of Test2" << endl; }

};

int main() {

try {

Test1 t1; // Constructed and destructed

Test2 t2; // Partially constructed

Test1 t3; // t3 is not constructed as this statement never gets executed

}

catch(int i) {

cout << "Caught " << i << endl;

}

return 0;

}

Output:

Constructing an Object of Test1

Constructing an Object of Test2

Destructing an Object of Test1

Caught 20

# Stack Unwinding in C++

The process of removing function entries from function call stack at run time is called Stack Unwinding.

In C++, when an exception occurs, the function call stack is linearly searched for the exception handler, and all the entries before the function with exception handler are removed from the function call stack.

So, exception handling involves Stack Unwinding if exception is not handled in same function (where it is thrown).

If there were some local class objects inside f1() and f2(), destructor for those local objects would have been called in Stack Unwinding process.

Stack Unwinding also happens in Java when exception is not handled in same function.

#include <iostream>

using namespace std;

// A sample function f1() that throws an int exception

void f1() throw (int) {

cout<<"\n f1() Start ";

throw 100;

cout<<"\n f1() End ";

}

// Another sample function f2() that calls f1()

void f2() throw (int) {

cout<<"\n f2() Start ";

f1();

cout<<"\n f2() End ";

}

// Another sample function f3() that calls f2() and handles exception thrown by f1()

void f3() {

cout<<"\n f3() Start ";

try {

f2();

}

catch(int i) {

cout<<"\n Caught Exception: "<<i;

}

cout<<"\n f3() End";

}

// A driver function to demonstrate Stack Unwinding process

int main() {

f3();

return 0;

}

Output:

f3() Start

f2() Start

f1() Start

Caught Exception: 100

f3() End

# Termination vs. Resumption

There are two basic models in exception-handling theory.

1. Termination
2. Resumption

**Termination** (which is what C++ supports) you assume the error is so critical there’s no way to get back to where the exception occurred. Whoever threw the exception decided there was no way to salvage the situation, and they don’t want to come back.

**Resumption** It means the exception handler is expected to do something to rectify the situation, and then the faulting function is retried, presuming success the second time. If you want resumption, you still hope to continue execution after the exception is handled, so your exception is more like a function call – which is how you should set up situations in C++ where you want resumption-like behavior (that is, don’t throw an exception; call a function that fixes the problem).

Although resumption sounds attractive at first, it isn’t quite so useful in practice. One reason may be the distance that can occur between the exception and its handler.

# Exception specification

void f() throw(toobig, toosmall, divzero); // With exceptions

void f(); // any type of exception may be thrown from the function

void f() throw(); // no exceptions are thrown from a function

## unexpected( )

If you throw an exception that isn’t in exceptions set, what’s the penalty? The special function unexpected( ) is called when you throw something other than what appears in the exception specification.

## set\_unexpected( )

unexpected( ) is implemented with a pointer to a function, so you can change its behavior. You do so with a function called set\_unexpected( ) which, like set\_new\_handler( ), takes the address of a function with no arguments and void return value.

Also, it returns the previous value of the unexpected( ) pointer so you can save it and restore it later.

To use set\_unexpected( ), you must include the header file <exception>.

// Basic exceptions

// Exception specifications & unexpected()

#include <exception>

#include <iostream>

#include <cstdlib>

#include <cstring>

using namespace std;

class Up {};

class Fit {};

void g();

void f(int i) throw (Up, Fit) {

switch(i) {

case 1: throw Up();

case 2: throw Fit();

}

g();

}

// void g() {} // Version 1

void g() { throw 47; } // Version 2

void my\_unexpected() {

cout << "unexpected exception thrown";

exit(1);

}

int main() {

set\_unexpected(my\_unexpected); // (ignores return value)

for(int i = 1; i <=3; i++)

try {

f(i);

}

catch(Up) {

cout << "Up caught" << endl;

}

catch(Fit) {

cout << "Fit caught" << endl;

}

return 0;

}

Output:

Up caught

Fit caught

unexpected exception thrown

* my\_unexpected( ) function has no arguments or return value, following the proper form for a custom unexpected( ) function.
* Your new unexpected( ) function must not return. However, it can throw another exception (you can even rethrow the same exception), or call exit( ) or abort( ).
* If unexpected( ) throws an exception, the search for the handler starts at the function call that threw the unexpected exception. (This behavior is unique to unexpected( ).)
* In the call to set\_unexpected( ), the return value is ignored, but it can also be saved in a pointer to function and restored later.
* In main( ), the try block is within a for loop so all the possibilities are exercised. Note that this is a way to achieve something like resumption.

# Uncaught exceptions

If none of the exception handlers following a particular try block matches an exception, that exception moves to the next-higher context.

If no handler at any level catches the exception, it is “uncaught” or “unhandled.” An uncaught exception also occurs if a new exception is thrown before an existing exception reaches its handler – the most common reason for this is that the constructor for the exception object itself causes a new exception.

## terminate( )

If an exception is uncaught, the special function terminate( ) is automatically called. Like unexpected( ), terminate is actually a pointer to a function. Its default value is abort( ), which immediately exits the program with no calls to the normal termination functions (which means that destructors for global and static objects might not be called).

No cleanups occur for an uncaught exception; that is, no destructors are called. If you don’t wrap your code (including, if necessary, all the code in main( )) in a try block followed by handlers and ending with a default handler (catch(...)) to catch all exceptions, then you will take your lumps. An uncaught exception should be thought of as a programming error.

## set\_terminate( )

You can install your own terminate( ) function using the standard set\_terminate( ) function, which returns a pointer to the terminate( ) function you are replacing, so you can restore it later if you want.

Your custom terminate( ) must take no arguments and have a void return value. In addition, any terminate( ) handler you install must not return or throw an exception, but instead must call some sort of program-termination function. If terminate( ) is called, it means the problem is unrecoverable.

Like unexpected( ), the terminate( ) function pointer should never be null.

// Use of set\_terminate()

// Also shows uncaught exceptions

#include <exception>

#include <iostream>

#include <cstdlib>

using namespace std;

void terminator() {

cout << "I'll be back!" << endl;

abort();

}

void (\*old\_terminate)() = set\_terminate(terminator);

class Botch {

public:

class Fruit {};

void f() {

cout << "Botch::f()" << endl;

throw Fruit();

}

~Botch() { throw 'c'; }

};

int main() {

try {

Botch b;

b.f();

}

catch(...) {

cout << "inside catch(...)" << endl;

}

return 0;

}

Output:

Botch::f()

I'll be back!

Runtime Errors:

Abort signal from abort(3) (SIGABRT)

class Botch not only throws an exception inside f( ), but also in its destructor. This is one of the situations that causes a call to terminate( ), as you can see in main( ). Even though the exception handler says catch(...), which would seem to catch everything and leave no cause for terminate( ) to be called, terminate( ) is called anyway, because in the process of cleaning up the objects on the stack to handle one exception, the Botch destructor is called, and that generates a second exception, forcing a call to terminate( ).

**Thus, a destructor that throws an exception or causes one to be thrown is a design error.**

# Rethrowing an exception

Fortunately, C++ provides a way to rethrow the exact same exception as the one that was just caught.

To do so, simply use the throw keyword from within the catch block (with no associated variable), like so:

#include <iostream>

class Base {

public:

Base() {}

virtual void print() { std::cout << "Base"; }

};

class Derived : public Base {

public:

Derived() {}

void print() override { std::cout << "Derived"; }

};

int main() {

try {

try {

throw Derived{};

} catch (Base& b) {

std::cout << "Caught Base b, which is actually a ";

b.print();

std::cout << '\n';

throw; // note: We're now rethrowing the object here

}

} catch (Base& b) {

std::cout << "Caught Base b, which is actually a ";

b.print();

std::cout << '\n';

}

return 0;

}

Output:

Caught Base b, which is actually a Derived

Caught Base b, which is actually a Derived

# noexcept specifier (since C++11)

The noexcept specifier defines a function as non-throwing.

To define a function as non-throwing, we can use the noexcept specifier in the function declaration, placed to the right of the function parameter list:

void doSomething() noexcept; // this function is non-throwing

**NOTE**

* noexcept doesn’t actually prevent the function from throwing exceptions or calling other functions that are potentially throwing. Rather, when an exception is thrown, if an exception exits a noexcept function, std::terminate will be called. And note that if std::terminate is called from inside a noexcept function, stack unwinding may or may not occur (depending on implementation and optimizations), which means your objects may or may not be destructed properly prior to termination.
* functions differing only in their exception specification can not be overloaded.

## noexcept specifier with a Boolean parameter

The noexcept specifier has an optional Boolean parameter. noexcept(true) is equivalent to noexcept, meaning the function is non-throwing. noexcept(false) means the function is potentially throwing. These parameters are typically only used in template functions, so that a template function can be dynamically created as non-throwing or potentially throwing based on some parameterized value.

## noexcept operator

The noexcept operator can be used inside functions. It takes an expression as an argument, and returns true or false if the compiler thinks it will throw an exception or not. The noexcept operator is checked statically at compile-time, and doesn’t actually evaluate the input expression.

void foo() {throw -1;}

void boo() {};

void goo() noexcept {};

struct S{};

constexpr bool b1{ noexcept(5 + 3) }; // true; ints are non-throwing

constexpr bool b2{ noexcept(foo()) }; // false; foo() throws an exception

constexpr bool b3{ noexcept(boo()) }; // false; boo() is implicitly noexcept(false)

constexpr bool b4{ noexcept(goo()) }; // true; goo() is explicitly noexcept(true)

constexpr bool b5{ noexcept(S{}) }; // true; a struct's default constructor is noexcept by default

The noexcept operator can be used to conditionally execute code depending on whether it is potentially throwing or not. This is required to fulfill certain exception safety guarantees, which we’ll talk about in the next section.

# Cleaning up

C++ exception handling guarantees that as you leave a scope, all objects in that scope whose constructors have been completed will have destructors called.

## Constructors

The general difficulty is allocating resources in constructors. If an exception occurs in the constructor, the destructor doesn’t get a chance to deallocate the resource. This problem occurs most often with “naked” pointers.

## Making everything an object

To prevent this, guard against these “raw” resource allocations by placing the allocations inside their own objects with their own constructors and destructors. This way, each allocation becomes atomic, as an object, and if it fails, the other resource allocation objects are properly cleaned up.

Templates are an excellent way to modify the above example:

// Naked pointers

#include <fstream>

#include <cstdlib>

using namespace std;

class Cat {

public:

Cat() { cout << "Cat()" << endl; }

~Cat() { cout << "~Cat()" << endl; }

};

class Dog {

public:

void\* operator new(size\_t sz) {

cout << "allocating a Dog" << endl;

throw int(47);

}

void operator delete(void\* p) {

cout << "deallocating a Dog" << endl;

::delete p;

}

};

class UseResources {

Cat\* bp;

Dog\* op;

public:

UseResources(int count = 1) {

cout << "UseResources()" << endl;

bp = new Cat[count];

op = new Dog;

}

~UseResources() {

cout << "~UseResources()" << endl;

delete []bp; // Array delete

delete op;

}

};

int main() {

try { UseResources ur(3); }

catch(int) { cout << "inside handler" << endl; }

}

Output

UseResources()

Cat()

Cat()

Cat()

allocating a Dog

inside handler

// Safe, atomic pointers

#include <fstream>

#include <cstdlib>

using namespace std;

// Simplified. Yours may have other arguments.

template<class T, int sz = 1> class PWrap {

T\* ptr;

public:

class RangeError {}; // Exception class

PWrap() {

ptr = new T[sz];

cout << "PWrap constructor" << endl;

}

~PWrap() {

delete []ptr;

cout << "PWrap destructor" << endl;

}

T& operator[](int i) throw(RangeError) {

if(i >= 0 && i < sz) return ptr[i];

throw RangeError();

}

};

class Cat {

public:

Cat() { cout << "Cat()" << endl; }

~Cat() { cout << "~Cat()" << endl; }

void g() {}

};

class Dog {

public:

void\* operator new[](size\_t sz) {

cout << "allocating an Dog" << endl;

throw int(47);

}

void operator delete[](void\* p) {

cout << "deallocating an Dog" << endl;

::delete p;

}

};

class UseResources {

PWrap<Cat, 3> Bonk;

PWrap<Dog> Og;

public:

UseResources() : Bonk(), Og() {

cout << "UseResources()" << endl;

}

~UseResources() {

cout << "~UseResources()" << endl;

}

void f() { Bonk[1].g(); }

};

int main() {

try { UseResources ur; }

catch(int) { cout << "inside handler" << endl; }

catch(...) { cout << "inside catch(...)" << endl; }

return 0;

}

Output:

Cat()

Cat()

Cat()

PWrap constructor

allocating an Dog

~Cat()

~Cat()

~Cat()

PWrap destructor

inside handler

# Standard exceptions

The following tables describe the standard exceptions:

|  |  |
| --- | --- |
| Exception | The base class for all the exceptions thrown by the C++ standard library. You can ask what( ) and get a result that can be displayed as a character representation. |
| logic\_error | Derived from exception. Reports program logic errors, which could presumably be detected before the program executes. |
| runtime\_error | Derived from exception. Reports runtime errors, which can presumably be detected only when the program executes. |

The iostream exception class ios::failure is also derived from exception, but it has no further subclasses.

The classes in both of the following tables can be used as they are, or they can act as base classes to derive your own more specific types of exceptions.

|  |  |
| --- | --- |
| **Exception classes derived from logic\_error** | |
| domain\_error | Reports violations of a precondition. |
| invalid\_argument | Indicates an invalid argument to the function it’s thrown from. |
| length\_error | Indicates an attempt to produce an object whose length is greater than or equal to NPOS (the largest representable value of type size\_t). |
| out\_of\_range | Reports an out-of-range argument. |
| bad\_cast | Thrown for executing an invalid dynamic\_cast expression in run-time type identification |
| bad\_typeid | Reports a null pointer p in an expression typeid(\*p). |

|  |  |
| --- | --- |
| **Exception classes derived from logic\_error** | |
| range\_error | Reports violation of a postcondition. |
| overflow\_error | Reports an arithmetic overflow. |
| bad\_alloc | Reports a failure to allocate storage. |

# Programming with exceptions

## Where exceptions are not warranted

### Not for asynchronous events

Asynchronous events: events that happen outside the scope of the program, and thus events the program cannot anticipate. C++ exceptions cannot be used to handle asynchronous events because the exception and its handler are on the same call stack. That is, exceptions rely on scoping, whereas asynchronous events must be handled by completely separate code that is not part of the normal program flow (typically, interrupt service routines or event loops).

### Not for ordinary error conditions

If you have enough information to handle an error, it’s not an exception. You should take care of it in the current context rather than throwing an exception to a larger context. Also, C++ exceptions are not thrown for machine-level events like divide-by-zero. It’s assumed these are dealt with by some other mechanism, like the operating system or hardware.

### You’re not forced to use exceptions

Some programs are quite simple, many utilities, for example. It is acceptable in these programs to use assert( ) or to print a message and abort( ) the program, allowing the system to clean up the mess, rather than to work very hard to catch all exceptions and recover all the resources yourself.

### New exceptions, old code

Another situation that arises is the modification of an existing program that doesn’t use exceptions. You may introduce a library that does use exceptions and wonder if you need to modify all your code throughout the program. Assuming you have an acceptable errorhandling scheme already in place, the most sensible thing to do here is surround the largest block that uses the new library (this may be all the code in main( )) with a try block, followed by a catch(...) and basic error message. You can refine this to whatever degree necessary by adding more specific handlers, but, in any case, the code you’re forced to add can be minimal.

You can also isolate your exception-generating code in a try block and write handlers to convert the exceptions into your existing error-handling scheme.

It’s truly important to think about exceptions when you’re creating a library for someone else to use, and you can’t know how they need to respond to critical error conditions.

## Typical uses of exceptions

### Always use exception specifications

The exception specification is like a function prototype: It tells the user to write exceptionhandling code and what exceptions to handle. It tells the compiler the exceptions that may come out of this function.

Sometimes the functions it calls produce an unexpected exception, and sometimes an old function that didn’t throw an exception is replaced with a new one that

does, and you’ll get a call to unexpected( ). Anytime you use exception specifications or call functions that do, you should create your own unexpected( ) function that logs a message and rethrows the same exception.

### Nest your own exceptions

If you create exceptions for your particular class, it’s a very good idea to nest the exception classes inside your class to provide a clear message to the reader that this exception is used only for your class. In addition, it prevents the pollution of the namespace. You can nest your exceptions even if you’re deriving them from C++ standard exceptions.

### Use exception hierarchies

Exception hierarchies provide a valuable way to classify the different types of critical errors that may be encountered with your class or library. This gives helpful information to users, assists them in organizing their code, and gives them the option of ignoring all the specific types of exceptions and just catching the base-class type.

### Multiple inheritance

If you need polymorphic behavior with both of those base classes. It turns out that exception hierarchies are a useful place for multiple inheritance because a base-class handler from any of the roots of the multiply inherited exception class can handle the exception.

## Catch by reference, not by value

If you throw an object of a derived class and it is caught by value in a handler for an object of the base class, that object is “sliced” – that is, the derived-class elements are cut off and you’ll end up with the base-class object being passed. Chances are this is not what you want because the object will behave like a base-class object and not the derived class object it really is (or rather, was – before it was sliced).

#include <iostream>

using namespace std;

class Base {

public:

virtual void what() { cout << "Base" << endl; }

};

class Derived : public Base {

public:

void what() { cout << "Derived" << endl; }

};

void f() { throw Derived(); }

int main() {

try { f(); } catch(Base b) { b.what(); }

try { f(); } catch(Base& b) { b.what(); }

return 0;

}

Output:

Base

Derived

because, when the object is caught by value, it is turned into a Base object (by the copy constructor) and must behave that way in all situations, whereas when it’s caught by reference, only the address is passed and the object isn’t truncated, so it behaves like what it really is, a Derived in this case.

Although you can also throw and catch pointers, by doing so you introduce more coupling – the thrower and the catcher must agree on how the exception object is allocated and cleaned up. This is a problem because the exception itself may have occurred from heap exhaustion. If you throw exception objects, the exception-handling system takes care of all storage.

# Don’t cause exceptions in destructors

Because destructors are called in the process of throwing other exceptions, you’ll never want to throw an exception in a destructor or cause another exception to be thrown by some action you perform in the destructor. If this happens, it means that a new exception may be thrown before the catch-clause for an existing exception is reached, which will cause a call to terminate( ).

This means that if you call any functions inside a destructor that may throw exceptions, those calls should be within a try block in the destructor, and the destructor must handle all exceptions itself. None must escape from the destructor.

# Overhead

Of course it costs something for this new feature; when an exception is thrown there’s considerable runtime overhead. This is the reason you never want to use exceptions as part of your normal flow-of-control, no matter how tempting and clever it may seem.

Exception handling also causes extra information to be put on the stack by the compiler, to aid in stack unwinding.

When the exception handler is finished, the exception objects are properly destroyed.

# References

<https://www.geeksforgeeks.org/c-plus-plus/#ExceptionHandling>

Chapter 20 Exceptions | <https://www.learncpp.com/>

<https://en.cppreference.com/w/cpp/language/exceptions>

<https://en.cppreference.com/w/cpp/language/noexcept_spec>

# END