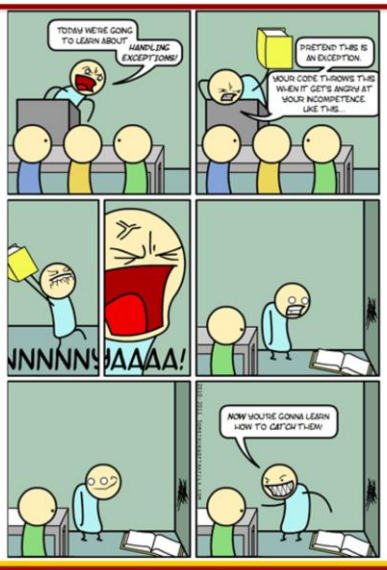
Exceptions and RTTI

ITP 435 – Spring 2016
Week 6, Lecture 1
Lecturer: Sanjay Madhav

USC Viterbi School of Engineering

University of Southern California

Exceptions



USC Viterbi School of Engineering

University of Southern California

http://somethingofthatilk.com/?id=202

(Basic) Exception Sample


```
try
{
   unsigned int size = 1000000000;
   int* i = new int[size];
}
catch (std::bad_alloc&)
{
   std::cout << "Memory allocation failed :(" << std::endl;
}
catch (...) // Avoid catch (...) when possible
{
   std::cout << "Unknown exception??" << std::endl;
}
```

Why I *Usually* Don't Like Exceptions

- This quote sums it up:

"In time-critical code, throwing an exception should *be* the exception, not the rule."

- Throwing exceptions triggers stack unwinding

USC Viterbi School of Engineering

University of Southern California

From boost article on exception handling

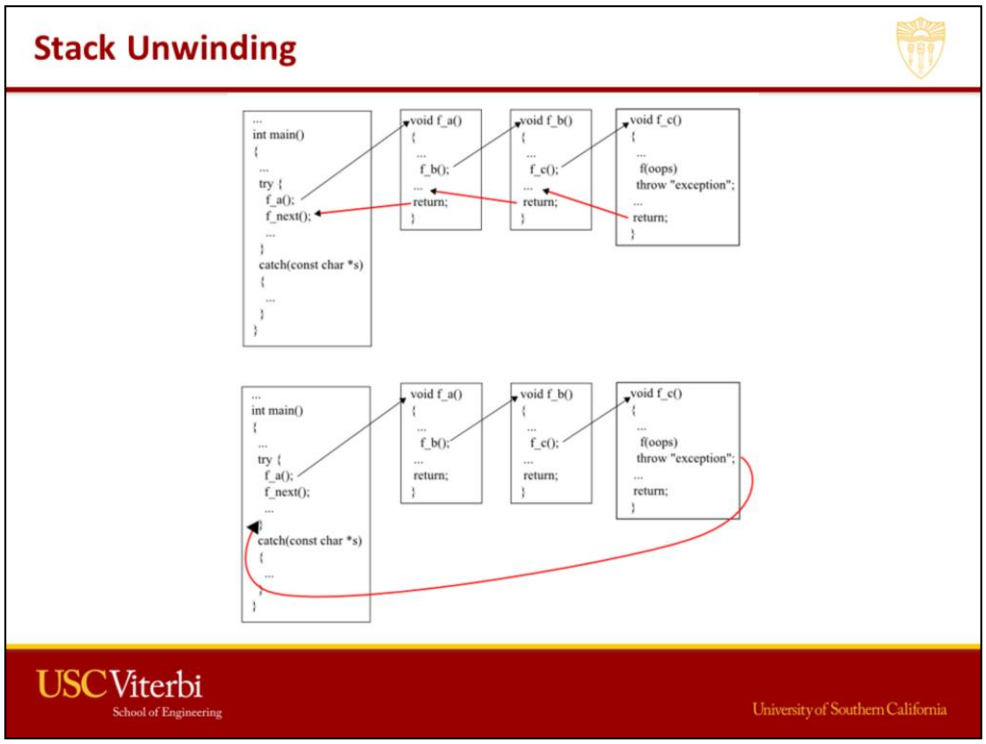


Diagram from http://www.bogotobogo.com/cplusplus/stackunwinding.php

More Stack Unwinding

- If an exception is thrown, the destructor of any classes local to the try block must be called in reverse order.

```
try
{
     ClassA A();
     ClassB B();
     ClassC C();
     throw;
}
```

- When the throw happens, C will be destructed, then B, then A.

But if you're going to use exceptions…

- Might as well use them right

What class type to throw?

- Never, ever, ever throw a class type whose constructor could also throw an exception.
- Example:

```
try
{
     throw std::string("BAD EXCEPTION HERE");
}
```

- What happens if string's constructor throws an exception?
- Answer: The program terminates

USC Viterbi School of Engineering

University of Southern California

How could std::string throw an exception? Out of memory!

More likely, it will lead to std::terminate which is the end of your program as you know it

std::exception

- Everything in the standard library throws an exception derived from `std::exception`...so you should also do this

```
#include <exception>
class MyException : public virtual std::exception
{
     // what is a virtual function which returns a description
     const char* what() const noexcept override { return "MY EXCEPTION!"; }
};
// later on...
try
{
     throw MyException();
}
catch (std::exception& e)
{
   std::cout << e.what() << std::endl; // "MY EXCEPTION!"
}
```

USC Viterbi School of Engineering

University of Southern California

Note the virtual inheritance. This is to protect ourselves from the diamond problem

noexcept

- If you have a function that doesn't throw an exception, specify `noexcept`

```
// Default: Can throw anything
void Function1();

// Should not throw an exception
void Function2() noexcept;
```

- NOTE: You have to specify `noexcept` on an `override` of "what"

catch (...)

- Problem with `catch (...)` is you could get crazy exceptions from the operating system ☹

- Added bonus of deriving from `std::exception`…
- You can avoid `catch (...)`
- Instead we can use `catch (std::exception& e)`

Unexpected Exceptions!

- What happens when you throw an exception from a `noexcept` function?

- `std::terminate`.

Exception-Safe Code

- Any code within a `try` block can't assume it succeeds.
- Therefore, you can't leave any dangling resources.
- Example from Effective C++:

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
     lock(&mutex);
     delete bgImage;
     ++imageChanges;
     bgImage = new Image(imgSrc);
     unlock(&mutex);
}
```

- What happens if `new` throws an exception?

A More In-depth Stack Unwinding Example


```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

A More In-depth Stack Unwinding Example


```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

Let's assume the entry point is `fa`

A More In-depth Stack Unwinding Example


```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

Calls fb…

A More In-depth Stack Unwinding Example


```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

Calls `fc`...

A More In-depth Stack Unwinding Example


```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

fc throws an exception.

Is it caught here?

A More In-depth Stack Unwinding Example


```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

fc throws an exception.

Is it caught here?

No! – So unwind the call stack

A More In-depth Stack Unwinding Example


```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

We're now back in ~~fb's~~ scope.

Before checking the catches, ***destruct*** myAA2!

A More In-depth Stack Unwinding Example


```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

Is it caught here?

A More In-depth Stack Unwinding Example


```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

Is it caught here?

No! – So unwind the call stack

A More In-depth Stack Unwinding Example


```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

We're now back in `fa`'s scope.

Before checking the catches, ***destruct*** `myAA`!

A More In-depth Stack Unwinding Example


```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

Is it caught here?

A More In-depth Stack Unwinding Example


```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

Is it caught here?

Yes! – because `ExceptThis` inherits from `std::exception`

Exception Summary

- If you're going to use exceptions, follow these rules:

1. Always throw an exception derived from `std::exception`
2. Never use `catch(...)`
3. Use `noexcept` when appropriate
4. Refactor code so it's exception-safe, as necessary

RTTI

- RTTI = ***Run-time Type Information*** (or Run-time Type Identification)
- In order for exceptions to work, C++ needs to figure out, at run-time, the type of the exception
- Thus RTTI and exceptions go hand in hand.

- RTTI ONLY works properly for classes that are polymorphic (eg. They have at least one virtual function, inherited or not).
- This is because RTTI information is stored in the virtual function table

dynamic_cast

- A down cast allows you to take a parent class pointer, and at runtime try to cast it to a child class.

- Eg. If you have a "Shape" pointer, and want to find out if it's a "Triangle" at runtime, you can do:

```
Shape* myShape;
Triangle* myTriangle = dynamic_cast<Triangle*> (myShape);
if (myTriangle) // dynamic_cast returns 0 if not a triangle
{
     // do something
}
```

- Should only be used in cases where you have a function you don't want to expose in the base class, which is rare.

typeid

- Allows you to figure out the type of something at runtime

```
class Person
{
public:
     virtual ~Person() {}
};
class Employee : public Person
{
};

// Later...
Person* ptr = new Employee();
if (typeid(*ptr) == typeid(Employee))
{
     // This is an employee!!
}
```

std::type_info

- The use of `typeid` returns a `std::type_info` class, which is defined in `<typeinfo>`

- Notable member functions:
  - `operator!=` (Self-explanatory)
  - `operator==` (Self-explanatory)
  - `name` (Returns implementation-specific name, as a const char*)

typeid().name()

- Returns a char* with the name of the type

```
// Outputs "class Employee"
// Note output string is compiler-dependant
std::cout << typeid(*ptr).name() << std::endl;
```

typeid gotcha

- If you use typeid on a *non-polymorphic* (eg. no virtual function) class, it wont' work as you might expect it to:

```
class Person
{
public:
//     virtual ~Person() {}
};
class Employee : public Person
{
};

// Later...
Person* ptr = new Employee();

// Outputs "class Person"
std::cout << typeid(*ptr).name() << std::endl;
```

Default RTTI Drawback

- Every single class with a virtual function has additional RTTI information stored in its virtual function table
- What happens if you have 10,000 classes with virtual functions, but you only need RTTI for 100 of them?
- Answer: Unnecessary memory waste
- Some C++ libraries (LLVM, for example) implement their own RTTI for this reason

USC Viterbi School of Engineering

University of Southern California

The default RTTI violates “you only pay for what you use”

Another reason is that the implementation of RTTI can vary based on compiler, so the usage characteristics can be somewhat nebulous.

LLVM RTTI – Why?

In an effort to reduce code and executable size, LLVM does not use RTTI (e.g. `dynamic_cast<>;`) or exceptions. These two language features violate the general C++ principle of "you only pay for what you use", causing executable bloat even if exceptions are never used in the code base, or if RTTI is never used for a class. Because of this, we turn them off globally in the code.

That said, LLVM does make extensive use of a hand-rolled form of RTTI that use templates like `isa<>`, `cast<>`, and `dyn_cast<>`. This form of RTTI is opt-in and can be added to any class. It is also substantially more efficient than `dynamic_cast<>`.

From the coding standards: http://llvm.org/docs/CodingStandards.html

Our Own Custom RTTI

- Step 1: Declare a “type info” class…

```
class TypeInfo {
public:
   // Takes pointer to super class' type info
   TypeInfo(const TypeInfo* super) : mSuper(super) {}
   // Is this type exactly matching the other pointer?
   bool IsExactly(const TypeInfo* other) const {
      return (this == other);
   }
   // Return the super type
   const TypeInfo* SuperType() const {
      return mSuper;
   }
private:
const TypeInfo* mSuper;
};
```

Our Own Custom RTTI, cont'd

- Step 2: For hierarchies that use it, declare the following:

```
class Object {
private:
   static const TypeInfo sType;
public:
   static const TypeInfo* StaticType() { return &sType; }
   virtual const TypeInfo* GetType() const { return &sType; }
};
```

- Initialize as follows (if base class):

```
const TypeInfo Object::sType(nullptr);
```

- A derived class would look like this:

```
const TypeInfo Derived::sType(SuperClass::StaticType());
```

Idea: Declare macros to make life easier…


```
#define DECL_OBJECT()  \
    private: \
    static const TypeInfo sType; \
    public: \
    static const TypeInfo* StaticType() { return &sType; } \
    const TypeInfo* GetType() const override { return &sType; } \


#define IMPL_OBJECT(d,s) \
    const TypeInfo d::sType(s::StaticType()); \
```

...then can use it like this


```
class Derived : public Object
{
   DECL_OBJECT();
};

IMPL_OBJECT(Derived, Object);
```

Is-A Function


```
// Returns true if ptr is-a Type
// Usage: IsA<Type>(ptr)
template <typename Other, typename This>
bool IsA(const This* ptr) {
   for (const TypeInfo* t = ptr->GetType();
        t != nullptr;
        t = t->SuperType()) {
      if (t->IsExactly(Other::StaticType())) {
         return true;
      }
   }
   return false;
}
```

Cast Function


```
// Casts ptr to Type if valid
// Usage: Cast<Type>(ptr)
template <typename Other, typename This>
Other* Cast(This* ptr) {
   if (IsA<Other>(ptr)) {
      return static_cast<Other*>(ptr);
   }
   else {
      return nullptr;
   }
}
```

Note

- Our RTTI implementation assumes single-inheritance…multiple inheritance gets messier

- This covers the approach used by LLVM (which does work for multiple inheritance):

http://llvm.org/docs/HowToSetUpLLVMStyleRTTI.html