Exceptions in C++

Technion - Israel Institute of Technology

Updated: November 2016
Outline

• Syntax
• Exceptions and Templates.
• Exceptions and Inheritance.
• Interface specification.
• Modifying the default mechanism.
• Resource acquisition.
A Simple Example

```c
int main(int argc, char *argv) {
    try {
        open_files(argv);
        allocate_memory();
        process();
        close_files();
    } catch (const char *msg) {
        cout << "Error: " << msg;
        return 1;
    }
    return 0;
}

void allocate_memory() {
    ...
    try {
        char *buff = new char[1L<<24];
    } catch (...) {
        throw "New failed!\n";
    }
}
template <class T> class Array {
public:
    explicit Array(int len_): len(len_), buff(new T[len]){}  // Error: Missing semicolon after braces.
~Array(void) { delete[] buff; }  // Error: Missing semicolon after braces.
int size(void) const { return len; }
T& operator[](int i) {
    if(i< 0 || i >= size())
        ???
        return buff[i];
}
const T& operator[](int i) const{
    return (*const_cast<Array<T> * const>this)[i];
}
private:
    const int len;
    T * const buff;
    Array(const Array &);
    Array & operator=(const Array &);
};
Error Handling in the Stack Example?

```cpp
template<class T> class Stack
{
    Array<T> a;
    size_t sp;

public:
    Stack(size_t n): sp(0), a(n) {}
    int empty(void) {
        return sp == 0;
    }
    int full(void) {
        return sp == a.size();
    }
    void push(T e) {
        if(full())
            ???
        a[sp++] = e;
    }
    T pop(void) {
        if(empty())
            ???
        return a[--sp];
    }
};
```
Classes used in Exceptions

• In many cases it is better to throw up the stack objects of a dedicated type.

```cpp
class Range {};  
class Overflow {};  
class Underflow {};  
```

Why?

• Classes with no members are legal in C++. An object of such a class consumes very little memory (but not 0).

```cpp
Why not size 0? Consider:
class X {};  
...
X a, b;  
X *pa=&a; X *pb=&b;  
if(pa==pb)  // the same object?  
```
template <class T> class Stack {
    ...

    public:
        ...
        void push(T e) {
            if (full())
                throw Overflow();
            buff[sp++] = e;
        }
        T pop(void) {
            if (empty())
                throw Underflow();
            return buff[--sp];
        }
};

Tip: Throw by value, Catch by reference
template<class T> class Array {
    const size_t N;
    T *buff;

public:
    Array(size_t n): N(n), buff(new T[N]) {}  
    ~Array(void) { delete[] buff; }
    T& operator[](size_t i) {
        if(i < 0 || i >= N)
            throw Range();
        return buff[i];
    }
};
Catching Array Errors

```cpp
void f(size_t n) {
    Array<int> v(n); // causes bad_alloc exception
    v[n + 1] = 7; // causes range exception
}
void g(size_t n) {
    ...
    try{
        // exceptions here are handled by
        ...
        // the handler defined below
        f(n); // might cause a range exception
        ...
    } catch(Range&) {
        error("Range error");
        return;
    } catch(bad_alloc&) {
        error("Memory allocation error");
        return;
    }
}
```
Syntax of the Try-Catch Construct

• The catch block must immediately follow the try block.
• There could be several catch blocks.
• The catch handlers are examined in the same order they appear in the function code.
• A function doesn't have to catch all exceptions. In fact, in many cases, it will not catch any.
• An uncaught exception will continue to unwind the stack. If no handler is found then the exception will terminate the program.
A possible implementation for a catch block:

```c
void g(size_t n){
    retry:
        try{
            Array<Complex> z(n);
            ...
        } catch(bad_alloc&){
            n--; // Free some memory
            ...
            goto retry;
        }
    ...
}
```

What else might we want to do?
struct Err {
    const double val;
    const char *func;
    Err(double v, char* f): val(v), func(f) {}}

double sqrt(double x) {
    if(x < 0)
        throw Err(x, __func__);...
}

void foo() {
    ...
    try{
        ...
    } catch(Err& e) {
        cerr << "Error in"<< e.func<< "with"<< e.val;
        ...
    }
    ...
}
Class specific exceptions are best expressed in terms of nested types:

```cpp
class Date {
    public:
        class Feb29Error {};
        Date(int d, int m, int y) {
            if (m == 2 && d == 29 && !leap(y))
                throw Feb29Error();
        }
    ...
};
```

```cpp
void f(int d, int m, int y) {
    Date today(d, m, y);
    ...
}
void g() {
    ...
    f(i1,i2,i3);
    ...
}
void h() {
    try{
        ...
        g();
        ...
    } catch(Date::Feb29Error&) {
        // Handle this error
    }
}
```
Template Exceptions

```
template<class T> class Stack {
    ...
    public:
        class Underflow {}
        class Overflow {}
        ...
    void push(T e) {
        if(full())
            throw Overflow();
        buff[sp++] = e;
    }
    T pop(void) {
        if(empty())
            throw Underflow();
        return buff[--sp];
    }
};
```

Throwing is the same as before. What’s the problem when catching?
There is no way to catch a template thrown object without specifying the parameters to the template (can you explain why?).
Exceptions and Inheritance

• Exceptions can be organized using inheritance hierarchy.

```cpp
struct StdlibError{};

struct NoMem: StdlibError{};
struct MathError: StdlibError{};
struct ZeroDivide: MathError{};
struct BadArcsinArg: MathError{};
struct BadLnArg: MathError{};
...
struct IoError: StdlibError{};
struct NetworkError: StdlibError{};
...
```

• Multiple inheritance can be also used.

```cpp
struct NetworkFileError: NetworkError, FileSystemError{};
```
Catching Derived Exceptions

• Handle exceptions from the most specific to the most general.

```cpp
try {
    ... 
} catch (ZeroDivide&) {
    ... 
} catch (BadArcsinArg&) {
    ... 
} catch (MathError&) {
    // Exception handler for all remaining math errors 
} catch (IoError&) {
    // Exception handler for I/O errors (that are not math errors) 
} catch (StdlibError&) {
    // Exception handler for all other standard library errors 
} catch (...) {
    // The syntax for catching all exceptions 
    // Exception handler for all other errors 
}
```
Catch Matching Algorithm

• A catch handler type (and optional variable) match the thrown object (static) type if:
  – Catch handler is ellipsis (...)
  – Catch handler type and thrown object type are identical
  – Handler type is a (visible) base class of the thrown object
  – Catch handler type is a pointer type, thrown object type is pointer, and standard pointer conversion can convert object to handler
    • The `void*` type in handler will catch all pointers
• If no match found, search goes on to next catch handler
• If no more catch handlers, stack is unwound to the next nesting block
• If no more try blocks program is terminated
What’s in a Catch Block?

• Header is just like a function prototype:
  – If parameter type is object of type T (by value), and the thrown object is of type T’ derived from T, then all extra information in thrown object is lost forever.
  – If references (or pointers) are used, then the dynamic type is preserved (and may be retrieved via RTTI)

```cpp
try {
    ...
} catch(SomeError e) {
    if(SomethingCanBeDone(e)) DoIt();
    else throw;
}
```

• Body is just like any C++ block:
  – Triggering exception is considered handled so re-throwing it will not cause an infinite loop.
  – A throw without arguments indicates a re-throw of the original exception thrown.
    • should only be done from within catch block (runtime error if not).
The errors which may occur inside a function are a part of its interface. The caller must be prepared for these.

A function may commit to throw only certain exceptions.

```cpp
double foo(double) throw(SomeException);
void bar(int) throw(); // No Exceptions!
void f() throw (SomeException1, SomeException2); // May throw exceptions of either type
```

If a function breaks its commitment then the exception handling mechanism calls `unexpected()`, which by default calls `terminate()` which by default calls `abort()`.

*This is deprecated in C++-11.*

You can declare a method does not throw exceptions using `noexcept`

Or not declare anything (the method can call any exception)

```cpp
void myFunc() noexcept;
```
The function `terminate()` is called in these situations:
- A thrown exception doesn’t find a handler
- `throw;` with no pending exception
- From inside `unexpected()`

Both `unexpected()` and `terminate()` are of the type `GlobHandler` defined by:

```c
typedef void (*GlobHandler)(void);
```

They can be changed by calling:
- The above return a pointer to previous handler

```c
GlobHandler set_unexpected(GlobHandler new_handler);
GlobHandler set_terminate(GlobHandler new_handler);
```

Don’t mess with all of this unless you are writing a library or an independent, large and complex subsystem.
int f(int n1, int n2, int n3) {
  char *buff1, *buff2, *buff3;
  if (0 == (buff1 = malloc(n1))) {
    fprintf(stderr, "Failed allocation of %d bytes\n", n1);
    return FAIL;
  }
  if (0 == (buff2 = malloc(n2))) {
    free(buff1);
    fprintf(stderr, "Failed allocation of %d bytes\n", n2);
    return FAIL;
  }
  if (0 == (buff3 = malloc(n3))) {
    free(buff2);
    free(buff1);
    fprintf(stderr, "Failed allocation of %d bytes\n", n3);
    return FAIL;
  }
  ...
  free(buff3); free(buff2); free(buff1);
  return DONE;
}
Robust Resource Acquisition: C++ Style

```cpp
Array::Array(size_t n): N(n), buff(new(nothrow) T[N]) {
    if(buff == NULL)
        throw Allocation();
}
Array::~Array(void) {
    delete[] buff;
}
void f(int n1, int n2, int n3) {
    Array<char> buff1(n1), buff2(n2), buff3(n3);
    ...
    return;
}
```

In a case of allocation failure `std::bad_alloc` will not be thrown, but null will be returned instead.

• The elegant implementation owes to:
  – Array template throwing an exception with allocation failure.
  – Exceptions call destructors of all created objects.
  – Destruction is in the opposite order of creation.
  – Exception handler prints error messages.
• Clearly, the unwinding of the stack is far from being trivial.
Exception Pitfalls

```cpp
template<class T> class Stack {
    ...

public:
    Stack& operator=(const Stack& s) {
        N = s.N;
        sp = s.sp;
        delete[] buff;
        buff = new(nothrow) T[s.N]; // T::T()
        if (buff == NULL)
            throw Allocation();
        for (int i = 0; i < sp; i++)
            buff[i] = s.buff[i]; // T::operator=(
    return *this;
}
    ...
};
```

- `sp` and `N` are assigned before the code that may fail, leaving the object in an inconsistent state!
- If `T::T()` throws an exception `buff` becomes a dangling reference, which will cause problems when the stack is destroyed!
Using RAII (e.g. smart pointer) for `buff` can prevent leaks and dangling pointers!
• If the loop fails the stack will not be destructed and the allocated buffer of \( T \) objects leaks!
• Unless buff is a smart pointer, whose destructor will be called even if the stack is not fully constructed.