

COMP322 - Introduction to C++

Lecture 10 - Overloading Operators and Exceptions

Dan Pomerantz

School of Computer Science

19 March 2012

Overloading operators in classes

It is useful sometimes to define *operators* to work on the new types that we are creating.

For example, in C++ or Java, we could define a type `ComplexNumber` to consist of

1. private property `real`
2. private property `complex`
3. `add()` method to add 2 `ComplexNumbers` together
4. `subtract()` method
5. etc

But this is not necessarily the most convenient to work with. Wouldn't it be nice to be able to add two numbers together with a `+`

Or what about being able to specify what `==` means? For example, if our type is complex, maybe we don't want to compare in the ordinary way.

Operator overloading

In Java, they make a philosophical decision that operators such as `+`, `-`, `=` can't be re-defined on Objects. In C++ though, we can do it!

```
const ComplexNumber ComplexNumber::operator+(const ComplexNumber &other)
{
    double real = real + other.real;
    double complex = complex + other.complex;
    return ComplexNumber(real, complex);
}
```

Now, we can do things like `ComplexNumber c = a + b;`

This is the sort of thing done in the string class with `+`. We can also overload `[]` if we wanted to make a class resembling vector for example or `=` if we wanted to specify copying or `==` to specify comparison.

Format

It always will look:

```
type operator sign (parameters) { definition }
```

operator means the text “operator” and sign means the symbol

<http://www.cplusplus.com/doc/tutorial/classes2/>

Motivation for exceptions

- ▶ Error handling is a difficult problem in general
- ▶ Organizing error codes and messages is tricky in C
- ▶ Error handling can lead to resource leaks and ugly code

```
bool f() { // true->success, false->failure
    int *pc = malloc(sizeof(int) * 100);
    if (pc == NULL) {
        return false;
    }
    FILE *fp = fopen(outfile, "w");
    if (fp == NULL) {
        free(pc); // release anything allocated
        return false;
    }
    // ...
    free(pc);
    fclose(fp);
    return true;
}
```

Motivation for exceptions, continued

- ▶ Using the “goto” statement is tempting:

```
bool f() {
    int *pc = NULL;
    FILE *fp = NULL;
    pc = malloc(sizeof(int)*100);
    if (pc == NULL) {
        goto error;
    }
    fp = fopen(outfile, "w");
    if (fp == NULL) {
        goto error;
    }
    // ...
    free(pc);
    fclose(fp);
    return true;

error:
    if (pc != NULL) free(fp);
    if (fp != NULL) fclose(fp);
    // ...
    return false;
}
```

What is an exception?

- ▶ A mechanism for handling *exceptional conditions*, including but not limited to errors.
- ▶ Exceptions are a mechanism for passing error information off to the runtime system, which can then select the appropriate handler for the error.
- ▶ Stroustrup: “One way of viewing exceptions is as a way of giving control to a caller when no meaningful action can be taken locally”.
- ▶ Alternative to printing messages or terminating programs within generic libraries.
- ▶ For C programmers, an exception is a safer, more flexible replacement for `setjmp()/longjmp()`.

Exception syntax in C++

C++ exception syntax is similar to that of Java:

- ▶ `try` - a “try” block associates a list of statements with one or more *exception handlers*.
- ▶ `catch` - one or more “catch” blocks follow the try block. These define the handler for a given type.
- ▶ `throw` - a “throw” statement passes the exception to the runtime system for delivery.
 - ▶ Control is immediately transferred to a handler associated with the nearest enclosing try block.
 - ▶ If no appropriate handler is found, the program exits.
 - ▶ The stack is “unwound” and destructors invoked as necessary.

A basic example

```
void g() {
    // etc.
    if (/* something goes wrong */) {
        throw 2;
    }
}

void f() {
    try {
        // ...
        g();
    }
    catch (int code) { // Handle int exceptions
        cerr << "Caught exception " << code << endl;
    }
    catch (...) { // Default handler
        cerr << "Caught unknown exception" << endl;
    }
}
```

Exceptions in C++ vs. Java

- ▶ C++ has no `finally` block
- ▶ C++ exceptions can throw *any* type
- ▶ C++ methods are never required to specify the exceptions they may throw

Some more details

The `catch` block must specify the type that is to be caught, it need not specify a parameter name.

If a parameter name is not specified, we can't examine the value of the exception or learn anything other than the type:

```
void f() {  
    try {  
        // ...  
    }  
    catch (int) { // Handle int exceptions anonymously  
        // deal with the exception  
    }  
    catch (...) { // Always anonymous, even the type is unknown  
    }  
}
```

Specifying exceptions for functions

- ▶ A function *may* specify the types of exceptions it throws.
- ▶ Other types, if thrown, will force an exit.
- ▶ No checking is done at compile time.

```
void f() { // No restrictions
    // ...
    throw 'c'; // OK
    throw 147; // OK
    throw string("oops!"); // OK
}

void g() throw() { // No exceptions
    // ...
    throw 2;      // Legal, but can't be caught
}

void h() throw(int, myexcept) { // May throw an int or 'myexcept'
    // ...
    throw 2;      // Can be caught
    throw 1.0;    // Can't catch a double exception
}
```

Nested exceptions

Try blocks can be nested within one another. The exception will be delivered to the innermost possible block:

```
void f() {  
    // ...  
    try {  
        // ...  
    }  
    catch (int e) {  
        try {  
            // complex recovery operation  
        }  
        catch (int e) {  
            // handler failed  
        }  
    }  
}
```

Nested exceptions and function calls

Exceptions can be delivered through multiple function calls:

```
void g() {
    // ...
    throw 13;
}

void f() {
    try {
        g();
    }
    catch (int e) { // Will be caught here...
        cerr << "f " << e << endl;
    }
}

int main() {
    try {
        f()
    }
    catch (int e) { // ...not here.
        cerr << "main " << e << endl;
    }
}
```

Exceptions and the stack

- ▶ A thrown exception will “unwind” the call stack.
- ▶ All fully-constructed objects that go out of scope are destroyed.
- ▶ Objects allocated with `new` are *not* destroyed.

```
void f() {  
    if (/* ... */) {  
        int *p = new int[100];  
        string s("a string");  
        // ...  
        throw 21; // s will be destroyed, p will not  
    }  
}
```

```
void g() {  
    try {  
        f();  
    }  
    catch (int e) {  
        // ...  
    }  
}
```

Exceptions within handlers

Exceptions thrown in a catch block must be caught in some higher enclosing handler, not in the current handler.

This code is legal and is not an infinite loop:

```
void f() {
    try {
        // ...
    }
    catch (int ec) {
        // ...
        throw 1;    // Would be handled in 'g'
    }
}

void g() {
    try {
        f();
    }
    catch (int ec) {
        // ...
    }
}
```


Re-throw

If your exception handler cannot completely handle the exception, it can “re-throw” the exception for the benefit of a caller:

```
void f() {  
    try {  
        // ...  
    }  
    catch (Exception & e) {  
        // ...  
        throw; // I've done all I can.  
    }  
}
```

The exception will be passed upwards. If you the exception is received by non-const reference or pointer, any modification will be passed to the next handler.

Exceptions and classes

Exceptions can use class types. These are generally preferred over built-in types, as it is easier both to organize exceptions and to pass useful information to handlers:

```
class Matherr { };
class Dividebyzero : public Matherr { };
class Overflow : public Matherr { };
class Underflow : public Matherr { };

void f() {
    try {
        // ...
    }
    catch (Dividebyzero) {
    }
    catch (Matherr) {
    }
}
```

Ordering of catch blocks

The order of exception handlers matters. When an exception occurs, C++ scans through the list of eligible exception handlers and selects the first one that is compatible. Therefore we often list catch blocks in order of increasing generality:

```
void f() {  
    try {  
        // ...  
    }  
    catch (Dividebyzero) { // Least general  
    }  
    catch (Matherr) { // More general  
    }  
    catch (...) { // Most general  
    }  
}
```

Exception hierarchies

In complex libraries or packages it may be useful to define one or more exception class hierarchies:

```
class Exception { // Base class of my exceptions
public:
    virtual string toString() = 0; // Convert information to string
};

class IOException: public Exception {
private:
    int code;
public:
    IOException(int c) { code = c; }
    virtual string toString() {
        ostringstream oss;
        oss << "I/O Error " << code << endl;
        return oss.str;
    }
};

void f() {
    //..
    throw IOException(42);
}
```

Polymorphic exceptions

In a hierarchy of exceptions, the same issues apply with assignment or passing of derived classes: data may be “sliced away” when a derived class is assigned to a base class.

We can avoid this by using either references (or pointers):

```
void f() {  
    try {  
        // ...  
    }  
    catch (Exception &e) {  
        cerr << e.toString();  
    }  
}
```

Passing exceptions by pointers is somewhat dangerous, as it may be unclear when and if to delete the exception.

Some exception guidelines

- ▶ A given try block is not required to catch all potential exceptions.
- ▶ While you can use any type in an exception, for larger programs it is probably a good idea to define a set of exception classes.
- ▶ Generally “catch by reference” is the norm.
- ▶ Don't throw exceptions in a destructor.
- ▶ The standard library may throw a number of possible exceptions; these are typically defined in `<stdexcept>`.
 - ▶ Standard hierarchy is rooted at `std::exception`
 - ▶ New exceptions commonly inherit from `std::runtime_error`

Exception safety

- ▶ Ideally, C++ code should go to some length to assure that it is *exception safe*.
 - ▶ Restore modified structures to consistent values
 - ▶ Release resources
- ▶ However, strong guarantees of exception safety are hard
- ▶ A standard design pattern helps maintain exception safety and generally results in simpler code.

Resource acquisition is initialization

- ▶ This is a basic pattern in C++, proposed by Bjarne Stroustrup.
- ▶ When objects are allocated on the stack, their destructor will be called when they go out of scope.
- ▶ We can use this to guarantee that resources are freed after either an exception or function return.
- ▶ Known as “resource acquisition is initialization” (RAII).

RAII - example

```
class infile {
private:
    FILE *m_file;
public:
    infile(string name) : m_file(fopen(name, "r")) {
        if (m_file == NULL) {
            throw runtime_error("can't open file");
        }
    }
    ~infile() {
        fclose(m_file);
    }
};

int f() {
    infile("readme.txt");

    // ...

    // We're guaranteed that if the fopen() succeeded, the
    // corresponding fclose() will occur!
}
```

RAII - definition

- ▶ Wherever possible, use local objects to manage resource acquisition, memory allocation, etc.
- ▶ The constructors and destructors of these objects are responsible for the actual acquisition or allocation.
- ▶ Explicitly construct contained objects in the initializer list.
- ▶ If these operations fail, the constructor should throw an exception.
- ▶ Careful exception handling in the constructor should allow it to restore the system to a valid state.

RAII - some details

- ▶ Once an object is fully constructed, it is guaranteed that its destructor will be called when the stack “unwinds”, whether because of an exception or normal return.
- ▶ Otherwise, the destructor will not be called.
- ▶ Constructors should clean up after themselves if necessary.

```
class A {
    B_ptr pb; // resource 1
    C_ptr pc; // resource 2
    A();
};

A::A() : pb(), pc() { // Use initializer list
    // if either elements' constructor throws an exception, the
    // object will not be constructed, and A's destructor will not be
    // called
}
```

Entire constructor as a try block

Often, it is useful to catch exception in the initializer list.

You can do this if you enclose an entire constructor in a try block:

```
Class::Class() try
  : x(0), y()
{
  // ...
}
catch(XErr &xe) {
  // trouble initializing 'x'
}
catch(YErr &ye) {
  // trouble initializing 'y'
}
```

Entire function as a try block

You can enclose an entire function body in a similar manner.

```
int g(int arg)
try {
    f(arg);
    return (0);
}
catch (Dividebyzero) {
    cerr << "Divide by zero\n";
    return arg+10; // Can return alternate values
}
catch (Matherr) {
    cerr << "Other math error\n";
    return arg+100; // Parameter is in scope
}
catch (...) {
    cerr << "Other...\n";
    return arg+1000;
}
```