CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 21
November 16, 2016

Operator Extensions

How to define operator extensions

Unary operator *`op`* is shorthand for `operator` *`op`* `()`.

Binary operator *`op`* is shorthand for `operator` *`op`* `(T arg2)`.

Some exceptions: Pre-increment and post-increment.

To define meaning of `++x` on type `T`, define `operator ++()`.

To define meaning of `x++` on type `T`, define `operator ++(int)` (a function of one argument). The argument is ignored.

Special case operator extensions

Some special cases.

- Subscript: `T& operator [](S index)`.
- Arrow: `X* operator ->()` returns pointer to a class `X` to which the selector is then applied.
- Function call; `T2 operator ()(`*`arg list`*`)`.
- Cast: `operator T()` defines a cast to type `T`.

Can also extend the `new`, `delete`, and `,` (comma) operators.

Virtue Demo

Virtual virtue

```
class Basic {
public:
    virtual void print(){cout <<"I am basic.  "; }
};
class Virtue : public Basic {
public:
    virtual void print(){cout <<"I have virtue.  "; }
};
class Question : public Virtue {
public:
    void print(){cout <<"I am questing.  "; }
};
```

Main virtue

What does this do?

```
int main (void) {
    cout << "Searching for Virtue\n";
    Basic* array[3];
    array[0] = new Basic();
    array[1] = new Virtue();
    array[2] = new Question();
    array[0]->print();
    array[1]->print();
    array[2]->print();
   return 0;
}
```

See demo `21a-Virtue`!

Exceptions

Exceptions

An *exception* is an event that prevents normal continuation.

Exceptions may be due to program errors or data errors, but they may also be due to external events:

- File not found.
- Insufficient permissions.
- Network failure.
- Read error.
- Out of memory error.

How to respond to different kinds of exceptions is application-dependent.

Exception handling

When an exception occurs, a program has several options:

- Try again.
- Try something else.
- Give up.

Problem: Exceptions are often detected at a low level of the code. Knowledge of how to respond resides at a higher level.

C-style solution using status returns

The C library functions generally report exceptions by returning status values or error codes.

Advantages: How to handle exception is delegated to the caller.

Disadvantages:

- Every caller must handle every possible exception.
- Exception-handling code becomes intermingled with the "normal" operation code, making program much more difficult to comprehend.

C++ exception mechanism

C++ exception mechanism is a means for a low-level routine to report an exception directly to a higher-level routine.

This separates exception-handling code from normal processing code.

An exception is reported using the keyword `throw`.

An exception is handled in a `catch` block.

Each routine in the chain between the reporter and the handler is exited cleanly, with all destructors called as expected.

Throwing an exception (demo `21b-Exceptions`)

`throw` is followed by an *exception* value.

Exceptions are usually objects of a user-defined exception type.

Example:

```
throw AgeError("Age can’t be negative");
```

Exception class definition:

```
class AgeError {
  string msg;
public:
  AgeError(string s) : msg(s) {}
  ostream& printError(ostream& out) const { return out<< msg; }
};
```

Catching an exception

A try region defines a section of code to be monitored for exceptions.
Immediately following are catch blocks for handling the exceptions.

```
try {
   ...  //run some code
}
catch (AgeError& aerr) {
    // report error
    cout<< "Age error: ";
    aerr.printError( cout )<< endl;
    // ... recover or abort
}
```

The catch parameter should generally be a reference parameter as in this example.

What kind of object should an exception throw?

`catch` filters the kinds of exceptions it will catch according to the type of object thrown.

For this reason, each kind of exception should throw it's own type of object.

That way, an exception handler appropriate to that kind of exception can catch it and process it appropriately.

While it may be tempting to throw a string that describes the error condition, it is difficult to process such an object except by printing it out and aborting (like `Fatal()`).

Properly used, exceptions are much more powerful than that.

Example: Stack template throws exception

It is an error to pop an empty stack.

We have given several sample stack implementations. Here's what they each do when attemption to pop an empty stack:

Demo	**Action on empty pop error**
`12-BracketsWithMove`	undefined (programmer must avoid)
`19-Virtual/linear.hpp`	return `nullptr`
`20b-Evaluate`	return `(T)0`
`21c-Exceptions-stack`	`throw` exception

See demo `21c-Exceptions-stack` for one way to handle an empty pop error using `throw`.

Polymorphic exception classes

A `catch` clause can catch polymorphic exception objects.

Demo `21d-Exceptions-cards` shows how this can be used to provide finer error control.

The base exception class `Bad` has a virtual print function. Derived from it are two classes `BadSuit` and `BadSpot`.

The catch clause `catch (bad& bs) {...}` will catch all three kinds of errors: bad suit, bad spot, and bad both.

These are errors that can arise while reading a playing card from the user.