C++ Virtual Methods, Error Handling, and Parsing

2501ICT/7421ICTNathan

René Hexel and Joel Fenwick

School of Information and Communication Technology
Griffith University

Semester 1, 2012

Outline

Review

- Lists and Arrays as part of the Standard API
  - → `NSArray` and `NSMutableArray` in Objective-C
  - → `vector` and `list` in C++
- Objective-C allows run-time reflection of Collection objects
  - → allows objects of multiple classes within a single collection
  - → container classes for primitive types
  - → `NSNumber`, `NSValue`, `NSNull`, and `NSData`
- C++ requires compile time templates
  - → allows only one type of element per collection
  - → e.g. `vector<int>`, `list<string>`, etc.
- C++ Namespaces and Operator Overloading
  - → save typing
  - → can make source code more readable
  - → need to be used with care!

Virtual Methods in C++

Virtual Methods in C++

Why Virtual Methods are needed – a Trick Question

Example (unexpectedly prints: `Classes:  A A `)

```
#include <iostream>
#include <vector>
using namespace std;

class A            { public:  void print() { cout << "A "; } };
class B: public A { public:  void print() { cout << "B "; } };

int main(int argc, char *argv[])
{
  A *a = new A;                                  // one instance of each class
  B *b = new B;
  vector<A *> vec(2);                             // a vector with two objects

  vec[0] = a;                                    // first object is a
  vec[1] = b;                                    // second object is b

  cout << "Classes:  ";

  vector<A *>::iterator e = vec.begin();          // enumerate vector
  while (e != vec.end())                         // for each element
         (*e++)->print();                         // invoke the print method

  cout << endl;

  return EXIT_SUCCESS;
}
```

Another Example

Let's look at the `Point` and `Point3D` classes again. Suppose `Point::getLength()` returns `x+y` and `Point3D::getLength()` returns `x+y+z`:

Example (unexpected results)

```
Point   *pa = new Point(1, 2);
Point3D *pb = new Point3D(1, 2, 3);
Point   *pc = pb;


pa->getLength() == 3
pb->getLength() == 6
pc->getLength() == 3
```

Observation

`pb` and `pc` point at the same object but the `getLength()` methods give different results.

This is different from Java and Objective-C.

$\Rightarrow$ If you want C++ to check which version of a method it should call, the method needs to be *virtual*.

Virtual Methods

- C++ is a static language
  - classes are determined at compile time
    - which method gets called also is determined at compile time
    - → even if object points to subclass `B`, superclass `A`'s `print()` method is invoked
  - → unlike Java and Objective-C
- Virtual Methods
  - actual method (parent or child class) is referenced within the class
  - → `virtual` keyword
    - also works for destructors: → destructors usually need to be virtual!
    - does not work for constructors → workaround: virtual factory methods

A Virtual Example

Example (now prints: `Classes:  A B `)

```
#include <iostream>
#include <vector>
using namespace std;

class A            { public:  virtual void print() { cout << "A "; } };
class B: public A { public:  virtual void print() { cout << "B "; } };

int main(int argc, char *argv[])
{
  A *a = new A;                                  // one instance of each class
  B *b = new B;
  vector<A *> vec(2);                             // a vector with two objects

  vec[0] = a;                                    // first object is a
  vec[1] = b;                                    // second object is b

  cout << "Classes:  ";

  vector<A *>::iterator e = vec.begin();          // enumerate vector
  while (e != vec.end())                         // for each element
         (*e++)->print();                         // invoke the print method

  cout << endl;

  return EXIT_SUCCESS;
}
```

Error Handling

Error Handling

C++ Error Handling

- Most error handling in C++ is also in-band
  - → return value indicates failure
    - e.g. `NULL` instead of a returned object, a boolean set to `FALSE`, an int set to `-1`, etc.
    - needs to be documented in the API
  - → requires explicit error checking
    - e.g. `if (object == NULL)` ...statements
- C++ method invocations on `NULL` objects are *unsafe*
  - → will cause your program to crash!

Exception Handling

- C++ uses `try`/`catch` for exception handling
  - → very similar to Java
- `try`
  - starts an exception handling domain
    - → like `NS_DURING` in Objective-C
    - → exceptions that occur will be caught
- `catch()`
  - the actual exception handler
  - catches exceptions that occur in the handling domain
    - → like `NS_DURING` in Objective-C
  - the parameter to `catch()` determines which type of exception gets caught
    - → multiple `catch()` blocks required for differentiating exceptions
    - → a `catch(...)` catches all remaining exceptions

C++ Exception Handling Example

Example (probably prints: `exception allocating vector`)

```
#include <vector>
#include <iostream>
#include <stdexcept>

using namespace std;

int main(int argc, char *argv[])
{
  try
  {
    vector<long long> *vec = new vector<long long>(500000000, 0);        // too big

    cout << vec->size() << endl;

    delete vec;
  }
  catch (...)
  {
    cout << "exception allocating vector"<< endl;

    return EXIT_FAILURE;
  }

  return EXIT_SUCCESS;
}
```

Throwing Exceptions

- `throw` statement
  - takes any valid C++ expresssion as parameter

  → can throw anything, not just `exception` objects

  → type used with `catch()` needs to match type in `throw`

Example

```
throw 42;                          // throw a constant integer of 42
throw some_variable;               // throw the value of some_variable
throw "An Exception";              // throw the C string "An Exception"
throw string("An Exception");      // throw the C++ string "An Exception"
```

C++ Exception Throwing Example

Example (prints: `MyException:  reason 42`)

```
#include <iostream>

void some_function(void)
{
  throw "MyException:  reason 42";                // raise a C string exception
}

int main(int argc, char *argv[])
{
  try
  {
    some_function();                              // call some function
  }
  catch (const char *localException)
  {
    std::cout << localException << std::endl;    // print exception
  }

  return EXIT_SUCCESS;
}
```

Input Parsing

Object-Oriented Input Parsing

Input Parsing Introduction

- We have seen how to print output in a formatted way
  - → `printf()` and `sprintf()` in C
  - → `NSLog()` and `+stringWithFormat:` in Objective-C
  - → `std::cout` in C++
- Parsing formatted input in C
  - → `scanf()` and `sscanf()`

→ How can formatted input be parsed in Objective-C and C++?

C++ File and Standard Input

- `getline(`*`input_stream`*`, `*`string`*`)`
  - read one line from *input_stream* into *string*
- `getline(`*`input_stream`*`, `*`string`*`, `*`character`*`)`
  - read from *input_stream* until *character* is encountered
- `cin`
  - an `istream` object that reads from `stdin`

  → `getline(cin, `*`string`*`);`
    - read a line from `stdin` into *string*

cin Example

Example (using cin)

```
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char *argv[])
{
  cout << "Enter your input:  ";                    // prompt user
  fflush(stdout);                                   // flush output

  string inputString;                               // the string to read

  getline(cin, inputString);                        // read one line

  cout << "You entered:  "<< inputString << endl;       // print user input

  return EXIT_SUCCESS;
}
```

Parsing Input

- » operator reads objects from a stream
  - destination object determines kind of data to be read
    - → `string` – scans a single word of text
    - → `int` – scans an integer
    - → `double` – scans a double
      - ...
- `istringstream`
  - a “stream” class for parsing strings
  - → uses the » in the same way as reading from a file

C++ Parsing Example

Example (prints: `Einstein, Albert was born in 1879`)

```
#include <iostream>
#include <string>
#include <sstream>                                  // use string streams

using namespace std;

int main(int argc, char *argv[])
{
  string input = "Albert Einstein, 1879";          // some input
  string firstName, lastName;                      // parsing variables
  int yearOfBirth;

  istringstream scanner(input);                    // create a scanner stream

  scanner >> firstName;                             // read the first word
  char c;
  do { c = scanner.get(); } while (isspace(c)); // skip white space
  scanner.putback(c);                              // put back non-space character
  getline(scanner, lastName, ',');                 // read until ','
  scanner >> yearOfBirth;                           // read an int

  cout << lastName << ", "<< firstName;             // print output
  cout << " was born in "<< yearOfBirth << endl;

  return EXIT_SUCCESS;
}
```