12. A bit of C++

Oscar Nierstrasz
Roadmap

> C++ vs C
> C++ vs Java
> References vs pointers
> C++ classes: Orthodox Canonical Form
> A quick look at STL — The Standard Template Library
Roadmap

- C++ vs C
- C++ vs Java
- References vs pointers
- C++ classes: Orthodox Canonical Form
- A quick look at STL — The Standard Template Library
Essential C++ Texts

What is C?

> C is a general purpose, procedural, imperative language developed in 1972 by Dennis Ritchie at Bell Labs for the Unix Operating System.
  — Low-level access to memory
  — Language constructs close to machine instructions
  — Used as a “machine-independent assembler”
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
    return 0;
}
My first C Program

#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
    return 0;
}
My first C Program

```
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
    return 0;
}
```

A preprocessor directive

Include standard io declarations
My first C Program

```c
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
    return 0;
}
```
My first C Program

```c
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
    return 0;
}
```

A preprocessor directive

Include standard io declarations

Write to standard output

char array
My first C Program

#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
    return 0;
}

A preprocessor directive
Include standard io declarations
Write to standard output
char array
Indicate correct termination

Friday, October 21, 11
What is C++?

A “better C” ([http://www.research.att.com/~bs/C++.html](http://www.research.att.com/~bs/C++.html)) that supports:

> Systems programming
> Object-oriented programming (*classes* & *inheritance*)
> Programming-in-the-large (*namespaces*, *exceptions*)
> Generic programming (*templates*)
> Reuse (large class & template libraries)
C++ vs C

Most C programs are also C++ programs.

Nevertheless, good C++ programs usually do not resemble C:

> avoid macros (use `inline`)
> avoid pointers (use references)
> avoid `malloc` and `free` (use `new` and `delete`)
> avoid arrays and `char*` (use `vectors` and `strings`)...
> avoid structs (use classes)

C++ encourages a different style of programming:

> avoid procedural programming
  — *model your domain* with classes and templates
Roadmap

> C++ vs C
> C++ vs Java
> References vs pointers
> C++ classes: Orthodox Canonical Form
> A quick look at STL — The Standard Template Library
package p2;

// My first Java program!
public class HelloMain {
    public static void main(String[] args) {
        System.out.println("hello world!");
        return 0;
    }
}

Hello World in Java
using namespace std;
#include <iostream>

// My first C++ program!
int main(void)
{
    cout << "hello world!" << endl;
    return 0;
}
“Hello World” in C++

Use the standard namespace

```cpp
using namespace std;
#include <iostream>
// My first C++ program!
int main(void)
{
    cout << "hello world!" << endl;
    return 0;
}
```
“Hello World” in C++

```
using namespace std;
#include <iostream>
// My first C++ program!
int main(void)
{
    cout << "hello world!" << endl;
    return 0;
}
```
“Hello World” in C++

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

// My first C++ program!
int main(void)
{
    cout << "hello world!" << endl;
    return 0;
}
```
"Hello World" in C++

```cpp
using namespace std;
#include <iostream>
// My first C++ program!
int main(void)
{
    cout << "hello world!" << endl;
    return 0;
}
```

- Use the standard namespace
- Include standard iostream classes
- A C++ comment
- cout is an instance of ostream

Friday, October 21, 11
“Hello World” in C++

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

// My first C++ program!
int main(void)
{
    cout << "hello world!" << endl;
    return 0;
}
```

- **Use the standard namespace**
- **Include standard iostream classes**
- **A C++ comment**
- **cout is an instance of ofstream**
- **operator overloading (two different argument types!)**
Makefiles / Managed Make in CDT

You could compile it all together by hand:

```
c++ helloWorld.cpp -o helloWorld
```
Makefiles / Managed Make in CDT

You could compile it all together by hand:

```
c++ helloWorld.cpp -o helloWorld
```

Or you could use a **Makefile** to manage dependencies:

```
helloWorld : helloWorld.cpp
c++ $@.cpp -o $@
```

```
make helloWorld
```
Makefiles / Managed Make in CDT

You could compile it all together by hand:

```
c++ helloWorld.cpp -o helloWorld
```

Or you could use a Makefile to manage dependencies:

```
helloWorld : helloWorld.cpp
c++ $@.cpp -o $@
```

Or you could use cdt with eclipse to create a standard managed make project
“C with Classes” designed by Bjarne Stroustrup in early 1980s:

> Originally a translator to C
  — Initially difficult to debug and inefficient

> Mostly *upward compatible* extension of C
  — “As close to C as possible, but no closer”
  — Stronger type-checking
  — Support for object-oriented programming

> Run-time efficiency
  — Language primitives close to machine instructions
  — *Minimal cost for new features*
## C++ Features

<table>
<thead>
<tr>
<th>Version</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C with Classes</strong></td>
<td>Classes as structs</td>
</tr>
<tr>
<td></td>
<td>Inheritance; virtual functions</td>
</tr>
<tr>
<td></td>
<td>Inline functions</td>
</tr>
<tr>
<td><strong>C++ 1.0 (1985)</strong></td>
<td>Strong typing; function prototypes</td>
</tr>
<tr>
<td></td>
<td>new and delete operators</td>
</tr>
<tr>
<td><strong>C++ 2.0</strong></td>
<td>Local classes; protected members</td>
</tr>
<tr>
<td></td>
<td>Multiple inheritance</td>
</tr>
<tr>
<td><strong>C++ 3.0</strong></td>
<td>Templates</td>
</tr>
<tr>
<td></td>
<td>Exception handling</td>
</tr>
<tr>
<td><strong>ANSI C++ (1998)</strong></td>
<td>Namespaces</td>
</tr>
<tr>
<td></td>
<td>RTTI (Runtime Type Information)</td>
</tr>
</tbody>
</table>
Java and C++ — Similarities and Extensions

Some Java Extensions:
> garbage collection
> standard abstract machine
> standard classes (came later to C++)
> packages (now C++ has namespaces)
> final classes
> autoboxing
> generics instead of templates
Java Simplifications of C++

> no pointers — just references
> no functions — can declare static methods
> no global variables — use public static variables
> no destructors — garbage collection and finalize
> no linking — dynamic class loading
> no header files — can define interface
> no operator overloading — only method overloading
> no member initialization lists — call super constructor
> no preprocessor — static final constants and automatic inlining
> no multiple inheritance — implement multiple interfaces
> no structs, unions — typically not needed
New Keywords

In addition to the keywords inherited from C, C++ adds:

<table>
<thead>
<tr>
<th>Exceptions</th>
<th>catch, throw, try</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declarations:</td>
<td>bool, class, enum, explicit, export, friend, inline, mutable, namespace, operator, private, protected, public, template, typename, using, virtual, volatile, wchar_t</td>
</tr>
<tr>
<td>Expressions:</td>
<td>and, and_eq, bitand, bitor, compl, const_cast, delete, dynamic_cast, false, new, not, not_eq, or, or_eq, reinterpret_cast, static_cast, this, true, typeid, xor, xor_eq</td>
</tr>
</tbody>
</table>

*(see [http://www.glenmccl.com/glos.htm](http://www.glenmccl.com/glos.htm))*
Roadmap

- C++ vs C
- C++ vs Java
- References vs pointers
- C++ classes: Orthodox Canonical Form
- A quick look at STL — The Standard Template Library
Memory Layout

**The address space consists of (at least):**

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Text:</strong></td>
<td>executable program text (not writable)</td>
</tr>
<tr>
<td><strong>Static:</strong></td>
<td>static data</td>
</tr>
<tr>
<td><strong>Heap:</strong></td>
<td>dynamically allocated global memory (grows upward)</td>
</tr>
<tr>
<td><strong>Stack:</strong></td>
<td>local memory for function calls (grows downward)</td>
</tr>
</tbody>
</table>

![Diagram showing memory layout with text, static, heap, and stack compartments]
Pointers in C and C++

```c
int i;
int *iPtr; // a pointer to an integer

iPtr = &i; // iPtr contains the address of i
*iPtr = 100;
```

<table>
<thead>
<tr>
<th>variable</th>
<th>value</th>
<th>Address in hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>100</td>
<td>456FD4</td>
</tr>
<tr>
<td>iPtr</td>
<td>456FD4</td>
<td>456FD0</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
A reference is an **alias** for another variable:

```c
int i = 10;
int &ir = i;  // reference (alias)
ir = ir + 1;  // increment i
```

*Once initialized, references cannot be changed.*

References are especially useful in **procedure calls** to avoid the overhead of passing arguments by value, without the clutter of explicit pointer dereferencing (\( y = *ptr; \))

```c
void refInc(int &n)
{
    n = n+1;  // increment the variable n refers to
}
```
References vs Pointers

*References should be preferred to pointers* except when:

> manipulating dynamically allocated objects
  — *new* returns an object pointer

> a variable must range over a set of objects
  — use a *pointer* to walk through the set
C++ Classes

C++ classes may be instantiated either *automatically* (on the stack):

```cpp
MyClass oVal; // constructor called
               // destroyed when scope ends
```

or *dynamically* (in the heap)

```cpp
MyClass *oPtr; // uninitialized pointer

oPtr = new MyClass; // constructor called
                    // must be explicitly deleted
```
# Constructors and destructors

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

using namespace std;

class MyClass {
private:
    string name;
public:
    MyClass(string name) : name(name) { // constructor
        cout << "create " << name << endl;
    }
    ~MyClass() {
        cout << "destroy " << name << endl;
    }
};
```
Constructors and destructors

```cpp
#include <iostream>
#include <string>
using namespace std;

class MyClass {
private:
    string name;
public:
    MyClass(string name) : name(name) {  // constructor
        cout << "create " << name << endl;
    }
    ~MyClass() {
        cout << "destroy " << name << endl;
    }
};
```

Include standard iostream and string classes
Constructors and destructors

```cpp
#include <iostream>
#include <string>
using namespace std;

class MyClass {
private:
    string name;
public:
    MyClass(string name) : name(name) {
        // constructor
        cout << "create " << name << endl;
    }
    ~MyClass() {
        cout << "destroy " << name << endl;
    }
};
```

Include standard iostream and string classes

Use initialization list in constructor
# Constructors and destructors

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

using namespace std;

class MyClass {
private:
    string name;
public:
    MyClass(string name) : name(name) { // constructor
        cout << "create " << name << endl;
    }
    ~MyClass() { 
        cout << "destroy " << name << endl;
    }
};
```

Include standard iostream and string classes

Use initialization list in constructor

Specify cleanup in destructor
Automatic and dynamic destruction

```cpp
MyClass& start() {
    MyClass a("a");  // automatic
    MyClass *b = new MyClass("b");  // dynamic
    return *b;  // returns a reference
}  // a goes out of scope

void finish(MyClass& b) {
    delete &b;  // need pointer to b
}  // need pointer to b
```
Automatic and dynamic destruction

MyClass& start() {
  MyClass a("a");
  MyClass *b = new MyClass("b");
  return *b;
}

void finish(MyClass& b) {
  delete &b;
}

#include "MyClass.h"
using namespace std;
int main (int argc, char **argv) {
  MyClass aClass("d");
  finish(start());
  return 0;
}
Automatic and dynamic destruction

```cpp
MyClass& start() {
    MyClass a("a");
    MyClass *b = new MyClass("b");
    return *b;
}

void finish(MyClass& b) {
    delete &b;
}

#include "MyClass.h"
using namespace std;
int main (int argc, char **argv) {
    MyClass aClass("d");
    finish(start());
    return 0;
}
```
Automatic and dynamic destruction

```cpp
MyClass& start() {
    MyClass a("a");
    MyClass *b = new MyClass("b");
    return *b;
}

void finish(MyClass& b) {
    delete &b;
}

#include "MyClass.h"
using namespace std;
int main (int argc, char **argv) {
    MyClass aClass("d");
    finish(start());
    return 0;
}
```
MyClass& start() {  
    MyClass a("a"); // automatic  
    MyClass *b = new MyClass("b"); // dynamic  
    return *b; // returns a reference (!) to b  
    // a goes out of scope
}

void finish(MyClass& b) {  
    delete &b; // need pointer to b
}

#include "MyClass.h"
using namespace std;

int main (int argc, char **argv) {
    MyClass aClass("d");
    finish(start());
    return 0;
}
Automatic and dynamic destruction

```cpp
MyClass& start() {  // returns a reference
    MyClass a("a");  // automatic
    MyClass *b = new MyClass("b");  // dynamic
    return *b;  // returns a reference (!) to b
    // a goes out of scope
}

void finish(MyClass& b) {  // need pointer to b
    delete &b;
}

#include "MyClass.h"
using namespace std;
int main (int argc, char **argv) {
    MyClass aClass("d");
    finish(start());
    return 0;
}
```

Friday, October 21, 11
Automatic and dynamic destruction

MyClass& start() {  
    MyClass a("a");  
    MyClass *b = new MyClass("b");  
    return *b;  
}  

void finish(MyClass& b) {  
    delete &b;  
}  

#include "MyClass.h"
using namespace std;

int main (int argc, char **argv) {  
    MyClass aClass("d");  
    finish(start());  
    return 0;  
}
Automatic and dynamic destruction

```cpp
MyClass& start() {
    MyClass a("a");
    MyClass *b = new MyClass("b");
    return *b;
}

void finish(MyClass& b) {
    delete &b;
}

#include "MyClass.h"
using namespace std;
int main (int argc, char **argv) {
    MyClass aClass("d");
    finish(start());
    return 0;
}
```

create d
create a
create b
destroy a
destroy b
destroy d

Friday, October 21, 11
Roadmap

- C++ vs C
- C++ vs Java
- References vs pointers
- C++ classes: Orthodox Canonical Form
- A quick look at STL — The Standard Template Library
Orthodox Canonical Form

Most of your classes should look like this:

```cpp
class myClass {
public:
    myClass(void); // default constructor
    myClass(const myClass& copy); // copy constructor
    ... // other constructors
    ~myClass(void); // destructor
    myClass& operator=(const myClass&); // assignment
    ... // other public member functions
private:
    ...
};
```
Why OCF?

If you don’t define these four member functions, C++ will generate them:

> **default constructor**
  — will call default constructor for each data member

> **destructor**
  — will call destructor of each data member

> **copy constructor**
  — will *shallow copy* each data member
  — pointers will be copied, not the objects pointed to!

> **assignment**
  — will *shallow copy* each data member
Example: A String Class

We would like a `String` class that protects C-style strings:

- strings are indistinguishable from `char` pointers
- string updates may cause memory to be corrupted

**Strings should support:**

- creation and destruction
- initialization from char arrays
- copying
- safe indexing
- safe concatenation and updating
- output
- length, and other common operations ...
class String
{
    friend ostream& operator<<(ostream&, const String&);

public:
    String(void); // default constructor
    ~String(void); // destructor
    String(const String& copy); // copy constructor
    String(const char*s); // char* constructor
    String& operator=(const String&); // assignment

    inline int length(void) const { return ::strlen(_s); }
    char& operator[](const int n) throw(exception);
    String& operator+=(const String&) throw(exception); // concatenation

private:
    char * _s; // invariant: _s points to a null-terminated heap string
    void become(const char*) throw(exception); // internal copy function
};
A Simple String.h

```cpp
class String {
    friend ostream& operator<<(ostream&, const String&);

public:
    String(void); // default constructor
    ~String(void); // destructor
    String(const String& copy); // copy constructor
    String(const char*s); // char* constructor
    String& operator=(const String&); // assignment

    inline int length(void) const { return ::strlen(_s); }
    char& operator[](const int n) throw(exception);
    String& operator+=(const String&) throw(exception); // concatenation

private:
    char * _s; // invariant: _s points to a null-terminated heap string
    void become(const char*) throw(exception); // internal copy function
};
```
A Simple String.h

```cpp
class String
{
friend ostream& operator<<(ostream&, const String&);
public:
    String(void); // default constructor
    ~String(void); // destructor
    String(const String& copy); // copy constructor
    String(const char*s); // char* constructor
    String& operator=(const String&); // assignment

    inline int length(void) const { return ::strlen(_s); } 
    char& operator[](const int n) throw(exception); 
    String& operator+=(const String&) throw(exception); // concatenation

private:
    char * _s; // invariant: _s points to a null-terminated heap string
    void become(const char*) throw(exception); // internal copy function
};
```
A Simple String.h

```cpp
class String
{
    friend ostream& operator<<(ostream&, const String&);
public:
    String(void);               // default constructor
    ~String(void);              // destructor
    String(const String& copy); // copy constructor
    String(const char* s);      // char* constructor
    String& operator=(const String&);       // assignment

    inline int length(void) const { return ::strlen(_s); }  
    char& operator[](const int n) throw(exception); 
    String& operator+=(const String&) throw(exception);     // concatenation

private:
    char * _s;  // invariant: _s points to a null-terminated heap string
    void become(const char*) throw(exception);    // internal copy function
};
```

A friend function prototype declaration of the String class

A friend function prototype declaration of the String class

Returns a reference to ostream

Operator overloading
class String
{
    friend ostream& operator<< (ostream&, const String&);

public:
    String (void); // default constructor
    ~String (void); // destructor
    String (const String& copy); // copy constructor
    String (const char* s); // char* constructor
    String& operator=(const String&); // assignment

    inline int length (void) const { return ::strlen(_s); }
    char& operator[] (const int n) throw (exception);
    String& operator+=(const String&) throw (exception); // concatenation

private:
    char* _s; // invariant: _s points to a null-terminated heap string
    void become(const char*) throw (exception); // internal copy function
};
A Simple String.h

```cpp
class String
{
    friend ostream& operator<<(ostream&, const String&);

public:
    String(void); // default constructor
    ~String(void); // destructor
    String(const String& copy); // copy constructor
    String(const char*s); // char* constructor
    String& operator=(const String&); // assignment

    inline int length(void) const { return ::strlen(_s); } // inline int length
    char& operator[](const int n) throw(exception); // operator[]
    String& operator+=(const String&) throw(exception); // concatenation

private:
    char * _s; // invariant: _s points to a null-terminated heap string
    void become(const char*) throw(exception); // internal copy function
};
```

A friend function prototype declaration of the String class

Returns a reference to ostream

Operator overloading

Operator overloading of =
Every constructor should *establish the class invariant*:

```cpp
String::String(void)
{
    _s = new char[1]; // allocate a char array
    _s[0] = '\0';    // NULL terminate it!
}
```

The *default constructor* for a class is called when a new instance is declared without any initialization parameters:

```cpp
String anEmptyString;    // call String::String()
String stringVector[10]; // call it ten times!
```
Default Constructors

Every constructor should *establish the class invariant*:

```c++
String::String(void)
{
    _s = new char[1]; // allocate a char array
    _s[0] = '\0';    // NULL terminate it!
}
```

Allocate memory for the string

The *default constructor* for a class is called when a new instance is declared without any initialization parameters:

```c++
String anEmptyString; // call String::String()
String stringVector[10]; // call it ten times!
```
Destructors

The String destructor must *explicitly free* any memory allocated by that object.

```cpp
String::~String (void) {
    delete [] _s;
}
```

*Every new must be matched somewhere by a delete!*

> use `new` and `delete` for *objects*

> use `new[]` and `delete[]` for *arrays!*
Our string copy constructor must create a *deep copy*:

```cpp
String::String(const String& copy)
{
    become(copy._s);        // call helper
}

void String::become(const char* s) throw (exception)
{
    _s = new char[:strlen(s) + 1];
    if (_s == 0) throw(logic_error("new failed"));
    ::strcpy(_s, s);
}
```

*From std*
A few remarks ...

> We **must** define a copy constructor, … else copies of Strings will *share the same representation*!  
   — Modifying one will modify the other!  
   — Destroying one will invalidate the other!

> We **must** declare copy as const, … else we won’t be able to construct a copy of a const String!  
   — Only const (immutable) operations are permitted on const values

> We **must** declare copy as String&, not String, … else a *new copy* will be made before it is passed to the constructor!  
   — Functions arguments are always passed by value in C++  
   — The “value” of a pointer is a pointer!

> The abstraction boundary is a class, *not an object*. Within a class, all private members are visible (as is copy._s)
Other Constructors

Class constructors may have arbitrary arguments, as long as their signatures are unique and unambiguous:

```cpp
String::String(const char* s) {
    become(s);
}
```

Since the argument is not modified, we can declare it as `const`. This will allow us to construct `String` instances from constant `char` arrays.
Assignment Operators

Assignment is different from the copy constructor because *an instance already exists*:

```cpp
String& String::operator=(const String& copy) {
    if (this != &copy) { // take care!
        delete [] _s;
        become(copy._s);
    }
    return *this; // NB: a reference, not a copy
}
```

> Return *String&* rather than *void* so the result *can be used in an expression*

> Return *String&* rather than *String* so the result *won’t be copied!*

> *this* is a pseudo-variable whose value is a pointer to the current object

—so *this* is the value of the current object, which is *returned by reference*
Implicit Conversion

When an argument of the “wrong” type is passed to a function, the C++ compiler looks for a constructor that will convert it to the “right” type:

```cpp
str = "hello world";
```

is implicitly converted to:

```cpp
str = String("hello world");
```

NB: compare to autoboxing in Java
Operator Overloading (indexing)

Not only assignment, but other useful operators can be “overloaded” provided their signatures are unique:

```cpp
char& String::operator[](const int n) throw(exception) {
    if ((n<0) || (length()<=n)) {
        throw(logic_error("array index out of bounds");
    }
    return _s[n];
}
```

NB: a non-const reference is returned, so can be used as an lvalue in an assignment.
### Overloadable Operators

The following operators may be overloaded:

<table>
<thead>
<tr>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>%</th>
<th>^</th>
<th>.</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>!</td>
<td>,</td>
<td>=</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;=</td>
<td>&gt;=</td>
</tr>
<tr>
<td>++</td>
<td>--</td>
<td>&lt;&lt;=</td>
<td>&gt;&gt;</td>
<td>==</td>
<td>!=</td>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>+=</td>
<td>-=</td>
<td>/=</td>
<td>%=</td>
<td>^=</td>
<td>&amp;=</td>
<td></td>
<td>=</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>&gt;&gt;=</td>
<td>[ ]</td>
<td>( )</td>
<td>-&gt;</td>
<td>-&gt;*</td>
<td>new</td>
<td>delete</td>
</tr>
</tbody>
</table>

**NB:** arity and precedence are fixed by C++
We would like to be able to write:

```cpp
    cout << String("TESTING ... ") << endl;
```

But:

- It can’t be a member function of `ostream`, since we can’t extend the standard library.
- It can’t be a member function of `String` since the target is `cout`.
- But it must have access to `String`’s private data

So ... we need a binary `function` `<<` that takes a `cout` and a `String` as arguments, and is a `friend` of `String`. 
We declare:

```cpp
class String {
    friend ostream&
        operator<<(ostream&, const String&);
    ...
};
```

And define:

```cpp
ostream&
operator<<(ostream& outStream, const String& s) {
    return outStream << s._s;
}
```
Roadmap

- C++ vs C
- C++ vs Java
- References vs pointers
- C++ classes: Orthodox Canonical Form
- A quick look at STL — The Standard Template Library
Standard Template Library

STL is a general-purpose C++ library of generic algorithms and data structures.

1. **Containers** store *collections of objects*
   — vector, list, deque, set, multiset, map, multimap

2. **Iterators** *traverse containers*
   — random access, bidirectional, forward/backward ...

3. **Function Objects** encapsulate *functions as objects*
   — arithmetic, comparison, logical, and user-defined ...

4. **Algorithms** implement *generic procedures*
   — search, count, copy, random_shuffle, sort, ... 

5. **Adaptors** provide an *alternative interface* to a component
   — stack, queue, reverse_iterator, ...
#include <iostream>
#include <stack>             // STL stacks
#include <string>            // Standard strings

void rev(void) {
    typedef stack<string> IOStack; // instantiate the template
    IOStack ioStack;              // instantiate the template class
    string buf;

    while (getline(cin, buf)) {
        ioStack.push(buf);
    }
    while (ioStack.size() != 0) {
        cout << ioStack.top() << endl;
        ioStack.pop();
    }
}
What we didn’t have time for ...

> virtual member functions, pure virtuals
> public, private and multiple inheritance
> default arguments, default initializers
> method overloading
> const declarations
> enumerations
> smart pointers
> static and dynamic casts
> Templates, STL
> template specialization
> namespaces
> RTTI
> ...
What you should know!

✎ What new features does C++ add to C?
✎ What does Java remove from C++?
✎ How should you use C and C++ commenting styles?
✎ How does a reference differ from a pointer?
✎ When should you use pointers in C++?
✎ Where do C++ objects live in memory?
✎ What is a member initialization list?
✎ Why does C++ need destructors?
✎ What is OCF and why is it important?
✎ What’s the difference between delete and delete[]?
✎ What is operator overloading?
Can you answer these questions?

- Why doesn’t C++ support garbage collection?
- Why doesn’t Java support multiple inheritance?
- What trouble can you get into with references?
- Why doesn’t C++ just make deep copies by default?
- How can you declare a class without a default constructor?
- Why can objects of the same class access each others private members?
Attribution-ShareAlike 3.0

You are free:
▪ to copy, distribute, display, and perform the work
▪ to make derivative works
▪ to make commercial use of the work

Under the following conditions:

**Attribution.** You must attribute the work in the manner specified by the author or licensor.

**Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

▪ For any reuse or distribution, you must make clear to others the license terms of this work.
▪ Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

http://creativecommons.org/licenses/by-sa/3.0/