Advanced C++

January 15, 2015

Mike Spertus

mike_spertus@symantec.com

Important Warning

- Read the following slides on optimized copy for culture. They are only to give you an eye on where we are heading and are *way* too hard to understand at this stage.
- Don't worry if it seems like a foreign language (you’re definitely not expected to understand it yet!). I promise it will be second nature by the end of the course.

Example: Copy

- How do we copy?
- In C

```
memcpy(cp, dp, n);
```

- This works in C++, but what if we are copying to/from an
  - array of objects?
  - list?
  - stream?
  - …

std::copy

- C++ standard library provides a standard copy function

```
copy(sp, sp+16, destp);
```

- From vector to array

```
vector<char> v;
...
copy(v.begin(), v.end(), destp);
```

std::copy—(Cont)

- From array to vector
- If we just copy to the vector, we will write off the end of the vector.

```
copy(cp, cp+8, v.end());// Wrong!
copy(cp, cp+8, back_inserter(v));
```

- The point is that copy takes "iterators," and std::back_inserter creates an iterator that appends to the end of a vector

Iterators

- Roughly speaking, iterators in C++ generalize pointers to array elements in C
- Much more general
  - Can iterate an array
  - Can iterate a container
  - Can iterate a stream
  - Can be input or output
  - Can be sequential or random access
  - Can append
  - etc.

Can we copy to a stream?

- Sure, we just need to turn it into an iterator
- Printing a comma-delimited vector of doubles

```
vector<double> v;
...
copy(v.begin(), v.end(),
 ostream_iterator<double>(cout,", "));
```

std::copy implementation

- Logically

```
template<class InItr, class OutItr>
OutItr
copy(InItr beg, InItr end, OutItr out)
{
  while(beg != end) {
    *out++ = *beg++;
  }
  return out;
}
```

Example: Optimized copy

- Programmers traditionally use their knowledge of underlying types to either write a memcpy or a hand-coded object-based copy loop. Breaking encapsulation like this makes the code less robust and extensible:
  - What happens when a programmer working on one part of the code adds a smart pointer to a struct definition without being aware that some other part of the program memcpy’s the struct
  - What happens when the array gets replaced by a vector?
  - What happens in generic code?

Active code

-
  ```
  char *cp1, *cp2, *cp3;
  T *tp1, *tp2, *tp3;
  vector<T> v;
  ...
  copy(cp1, cp2, cp3); // Should memcpy
  // The following should all either memcpy or
  // object copying as appropriate
  copy(tp1, tp2, tp3);
  copy(tp1, tp2, v.begin());
  copy(v.begin(), v.end(), tp3);
  ```
- The above code should morph appropriately if the definition of `T` changes. This would simultaneously optimize ease-of-use, maintainability, and performance

Templates to the rescue

- C++, Java, and .NET have generics, but they are very different, so don't be confused.
- In Java and .NET, templates are designed to create typesafe algorithms that are the same for all types
- C++ can do this too:

```
template<class InputIter, class OutputIter>
void copy(InputIter scan, InputIter end,
OutputIter out) {
    while(scan != end) {
          *out++ = *scan++;
    }
}
```

Overloading

- Let’s create another function named copy that just works in arrays of characters (so its safe to memcpy)

-
  ```
  void
  copy(const char *scan, const char *end, char *out) {
          memcpy(out, scan, end - scan);
  }
  ```
- Now `memcpy` is automatically called for character arrays because the copy on this slide is a more precise fit than the template function on slide 37
  - Client code still runs fine if types change to something else
  - Can’t do this with C# and Java because no overloading
- What about `long`, `const unsigned int`, `bool`, `double`, `char **`…?
  - Could address this with a lot of tedious code and ugly macros.
  - Easy to make a mistake
  - What if you have structs that are safe to `memcpy`?

The Mental Model

- Templates are the compile-time equivalent of object-oriented dispatch
- Rough mental model:

Run-time	Polymorphism	Inheritance	Virtual Overrride
Compile-time	Templates	(Unspecialized) Template	Template specialization

- Template specializations as being as important as virtual functions

Type traits

- Type traits are a set of template types introduced in C++11 (prior to that, they were available in boost) that inherit from the special types `true_type` or `false_type`.
- For example, `is_pointer` tells if a type is a pointer. If you are already familiar with template specializations, you can see how it is implemented below:

```
template <typename T> struct is_pointer : public false_type{};
template <typename T> struct is_pointer<T*> : public
true_type{};
```

Testing for "binary-copyability"

- We are going to use the type trait `is_trivially_copy_constructible`
- `is_trivially_copy_constructible<T>` inherits from `false_type` if nothing is known about T.
- `is_trivially_copy_constructible<T>` has specializations inheriting from `true_type` for all built-in type expressions that can be assigned with a binary copy.
- That's just what we need!

The code

- **Remember! Just listing for culture, completeness, and to start to familiarize you with the concepts. It is expected that there will be much that is not comprehensible in this listing at this stage of the course, although it will become second nature over the next ten weeks.**

```
template<typename I1, typename I2>
inline I2 optimized_copy(I1 first, I1 last, I2 out)
{
  typedef typename std::iterator_traits<I1>::value_type value_type;
   return copy_imp(first, last, out,
                   is_trivially_copy_constructible<value_type>());
}
```

The Code—Continued

```
// Remember: This code can and should seem like Greek now
// I promise it will become completely clear as the course proceeds
template<typename I1, typename I2, bool b>
I2
copy_imp(I1 first, I1 last, I2 out,
         const integral_constant<bool, b>&)
{
   while(first != last) {
      *out++ = *first++;
   }
   return out;
}

template<typename T>
T* copy_imp(T* first, T* last, T* out, const true_type&)
{
   memcpy(out, first, (last-first)*sizeof(T));
   return out+(last-first);
}
```

Is it worth it?

- The user interface is as simple and uniform as possible
- Type implementations are completely encapsulated
  - Changes to types don’t cause non-local problems
- Performance?

Version	Type	Time
Conventional	char	8s
Optimized	char	1s
Conventional	int	8s
Optimized	int	2.5s

The downside

- The code for `copy` is more complex
- However,…
  - All the complexity is in one place
  - The client code is less complex
  - Client programs will be more flexible and maintainable
- My opinion
  - If performance doesn't matter, then not justified
    - But don't put memcpy in your code either. All the advantages of preferring copy remain
  - If it does matter, then the performance gains may be worth it
  - If performance is critical, it may be worth providing the `has_trivial_assign` specialization for your classes, but only if benchmarking identifies this as a botttleneck

MAIN LECTURE

Creating our own types

- If we could only use a language' built-in types and couldn't define any of our own, the power of the language would be very much restricted to what was built-in
- In my solution to the Pascal's triangle problem, I sort of created my own Row and Triangle types
  - `typedef vector<int> Row;`
  - But what `typedef` does is let you give a new name to an existing type, so this is useful, but doesn't create genuinely new types

Classes

- To create your own type in C++, you must define a class
- Consider the following example (p. 61 in Koenig and Moo)

```
struct Student_info {
  string name;
  double midterm, final;
  vector<double> homework;
};  // Semicolon is required!
```

- See the sample “Grading” programs on chalk
- Unlike C, no need for:

```
typedef struct Student_info Student_info;
```

Hey, that's a `struct`, not a `class`!

- That's OK, the only difference between a struct and a class in C++ is different default visibility of members.
- The following is equivalent

```
class Student_info {
public:
   string name;
   double midterm, final;
   vector<double> homework;
};
```

Visibility of members

- public members are visible to everyone
- Protected members are visible to subclasses
- Private members are only visible to the class

Visibility (cont)

```
class A {
 void f() {
    cout << pub; // OK
    cout << prot; // OK
    cout << priv; // OK
  }
public:
  int pub;
protected:
  int prot;
private:
  int priv;
};
class B : public A {
  void g() {
    cout << pub; // OK
    cout << prot; // OK
    cout << priv; // Error
  }
};

void h(A a)
{
    cout << a.pub; // OK
    cout << a.prot; // Error
    cout << a.priv; // Error
}
```

Let’s add methods (p. 157)

```
struct Student_info { // In header
  string name;
  double midterm, final;
  vector<double> homework;

  istream& read(istream&);
  double grade() const;
};
// In .cpp file
double Student_info::grade() const
{
  return (midterm + final + median(homework))/3;
}
```

This is the same as the last slide

```
Struct Student_info { // In header
  string name;
  double midterm, final;
  vector<double> homework;

  istream& read(istream&);
  double grade() const {
    return (midterm + final + median(homework))/3;
  } // No semicolon!
}; // Semicolon!
```

Adding multiple grading strategies

```
struct Abstract_student_info { // In header
  string name;
  double midterm, final;
  vector<double> homework;

  istream& read(istream&);
  // Don’t define grading strategy here
  virtual double grade() const = 0;
};
```

Here are a couple of strategies

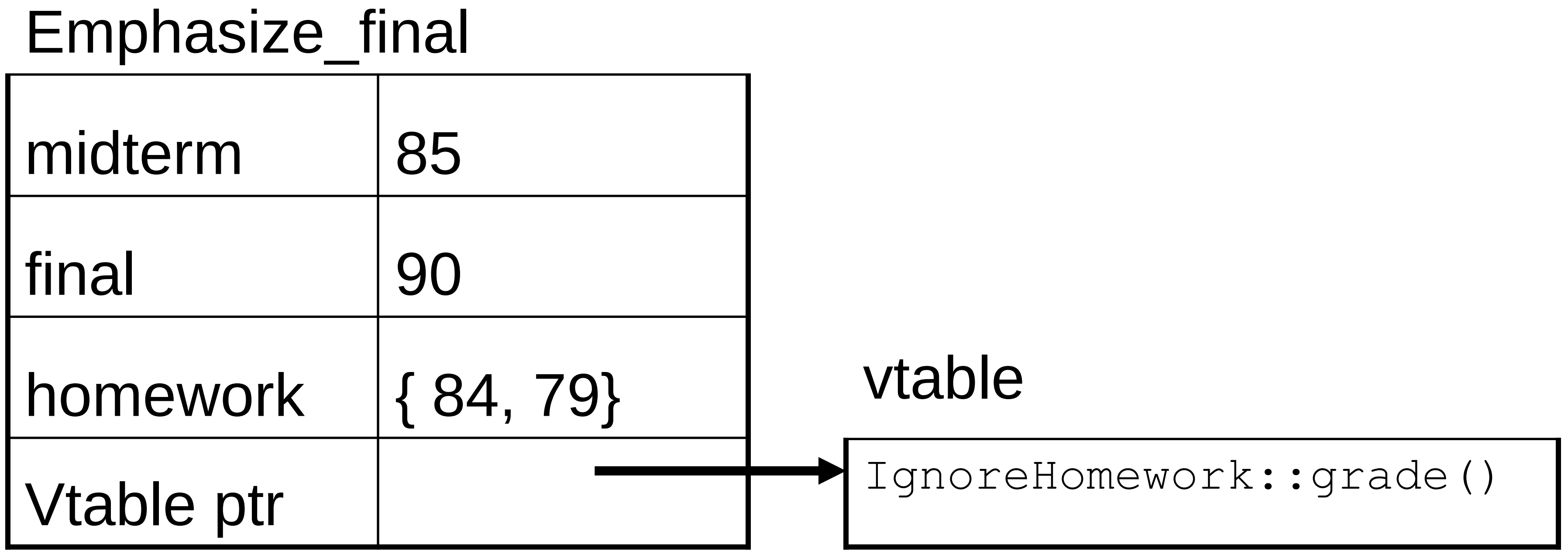
```
struct BalancedGrading: public Abstract_student_info {
  virtual double grade() const {
    return (midterm + final + median(homework))/3;
  }
};


struct IgnoreHomework: public Abstract_student_info {
  double grade() const {
    return (midterm + final)2; // Ignore the HW
  }
};
```

How do we choose which grading strategy to use?

```
int main()
{
  Abstract_student_info *si = new Balanced_grading();
  si->read(cin);
  cout << "Grade is " << si->grade() << endl;
  delete si; // We'll learn more about new
             // and delete later
  return 0;
}
```

Virtual function implementation



Emphasize_final

midterm	85
final	90
homework	{ 84, 79}
Vtable ptr	

vtable

```
IgnoreHomework::grade()
```

Static vs Dynamic types

- The static type is the type of the expression
  - Known at compile time
- The dynamic type is the type of the actual object referred to by the expression
  - Known only at run-time
- Static and dynamic type only differ due to inheritance

Static type vs Dynamic type

```
int i = 5; // S = int, D = int
Gorilla g; // S = Gorilla, D = Gorilla
Animal *a = new Gorilla // S = An, D = Gor
Animal a2 = *a; // Error!
```

Virtual vs. non-virtual method

- A virtual method uses the dynamic type
- A non-virtual method uses the static type

Virtual vs. Non-virtual method

```
struct Animal {
  void f()
    { cout << "animal f"; }
  virtual void g()
    { cout << "animal g"; }
};
struct Gorilla : public Animal{
  void f()
    { cout << "gorilla f"; }
  void g()
    { cout << "gorilla g"; }
};
Animal *a = new Gorilla;
a->f();
a->g();
```

Virtual method implementation/performance

- Since the compiler doesn’t know what the type is at compile-time, a virtual function is called through a pointer to a table of functions that is stored in the object
- For a non-virtual method, the compiler knows what method will be called and calls it directly
- In general, this only adds a couple of clock cycles, so the cost of making a function virtual is usually negligible
- But that is not the whole story

Benchmark 1

```
#include <iostream>
#include <math.h>
#include <boost/progress.hpp>
using namespace std;
class A {
public:
   // 15.83s if virtual 15.82 if not virtual on some compilers
   // 12s if virtual vs. 6 if not virtual on VS2013!
   virtual int f(double i1, int i2) {
       return static_cast<int>(i1*log(i1))*i2;
   }
};
int main()
{
   boost::progress_timer t;
   A *a = new A();
   int ai = 0;
   for(int i = 0; i < 100000000; i++) {
           ai += a->f(i, 10);
   }
   cout << ai << endl;
}
```

Benchmark 2:

```
#include <iostream>
#include <cmath>
#include <boost/progress.hpp>
using namespace std;
class A {
public: // 15.36s if virtual 0.22 if not virtual
   virtual int f(double d, int i) {
       return static_cast<int>(d*log(d))*i;
   }
};
int main()
{
   boost::progress_timer t;
   A *a = new A();
   int ai = 0;
   for(int i = 0; i < 100000000; i++) {
           ai += a->f(10, i);
   }
   cout << ai << endl;
}
```

What happened?

- The main performance cost of virtual functions is the loss of inlining and function level optimization
  - Not the overhead of the indirection
  - In the second benchmark, 10*log(10) only needed to be calculated once in the non-virtual case.
- Usually not significant, but this case is good to understand
  - The more virtual functions you use, the less the compiler can understand your code to optimize it

Signatures of virtual methods

- Usually, when overriding a virtual method, the overriding method has the same signature as the method in the base class
- What happens if it is different?

Constructors

- `new Student_info()` leaves `midterm`, `final` with nonsense values. (Use the original version. The one with the “pure virtual” method can’t be new’ed!)
- But not `homework`! We’ll understand that momentarily
- Fix as follows:

```
struct Student_info {
  Student_info() : midterm(0), final(0) {}
};
```

Destructors

- Consider

```
struct Use_grading_machine
 : public Abstract_student_info {
  Use_grading_machine() { // Constructor
    grading_machine = new Grading_machine();
  }
  // Use grading machine
  ~Use_grading_machine() {
    delete grading_machine;
  }
protected:
  Grading_machine *grading_machine;
};
```

Right idea, but not right

```
int main()
   {
     Abstract_student_info *si = new UseGradingMachine();
     si->read(cin);
     cout << "Grade is " << si->grade() << endl;
     delete si; // Calls Abstract_student_info’s destructor
     return 0;
   }
```

What we really need is for the destructor to be virtual!

```
Struct Abstract_student_info { // In header
  string name;
  double midterm, final;
  vector<double> homework;
  virtual ~Abstract_student_info() {}
  istream& read(istream&);
  // Don’t define grading strategy here
  virtual double grade() const = 0;
};
```

- Best practice: Classes that are designed to be inherited from should have virtual destructors

Review of when to inherit

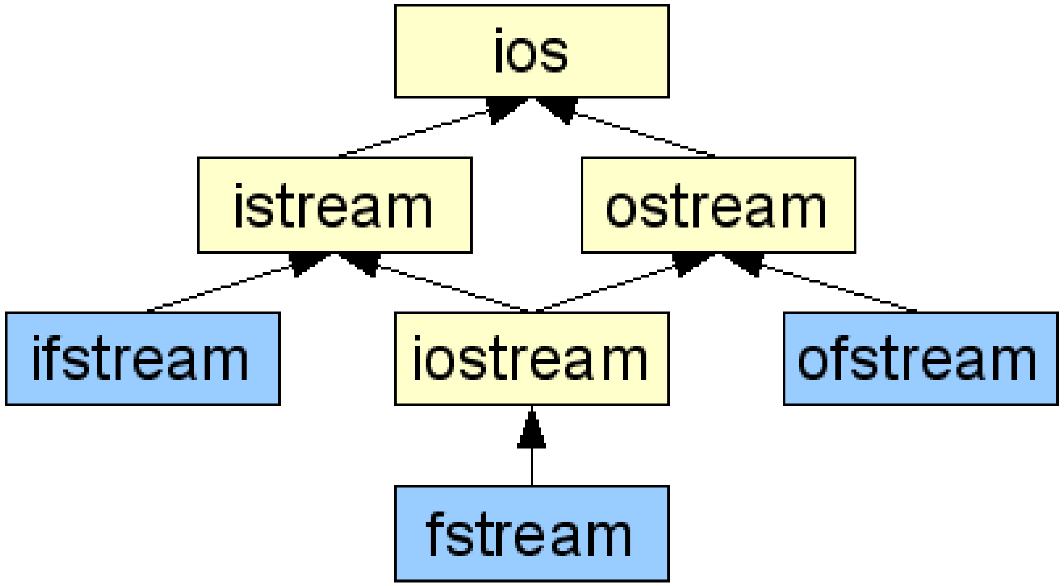
- Inherit to represent "is-a" relationship
  - A dachsund is a dog
- Use members to represent "has-a" relationship
  - A dachsund has a fur color
- Most languages don't allow multiple inheritance
  - Shouldn't an iostream, be both an istream and an ostream
  - In C++, it can be

Single inheritance

- class A : public B {…};
- class A : protected B {…};
- class A : private B {…};

Multiple inheritance

- For a good discussion, see http://www.phpcompiler.org/doc/virtualinheritance.html



Recommended reading

- All reading is from Koenig and Moo
- As of week 2, we have covered chapters 1-4, 5.1-5.3, 9, 13. If you need additional reinforcement, be sure to review those chapters.
- In week 3, we will be covering chapters 5-8, 14. You are encouraged to read those in advance

HW 2.1 (Part 1)

- In addition to std::copy, there is a similar function in the <algorithm> header called std::transform, which lets you apply a function to each element before copying. Look up or Google for std::transform to understand the precise usage.
- Write a program that puts several floats in a vector and then uses std::transform to produce a new vector with each element equal to the square of the corresponding element in the original vector and print the new vector (If you use ostream_iterator to print the new vector, you will likely get an extra comma at the end of the output. Don't worry if that happens).

HW 2.1 (Part 2)

- We will extend the program in part 1 to calculate and print the distance of a vector from the origin.
- There is also a function in the <numeric> header called `accumulate` that can be used to add up all the items in a collection.
  - (Googling for "accumulate numeric example" gives some good examples of using `accumulate`. We're interested in the 3 argument version of `accumulate`).
- After squaring each element of the vector as in part 1, use accumulate to add up the squares and then take the square root. (That this is the distance from the origin is the famous Pythagorean theorem, which works in any number of dimensions).

HW 2.1 (Extra credit part 3)

- There is also a four argument version of `accumulate` that can combine `transform` and `accumulate` in a single step. Use this to provide a more direct solution to Part 2 of this problem
    - In real life, you'd probably use the `inner_product` function in the `<numeric>` header, but don't do that for this exercise.

HW2.2

- One of the above slides referred to a function median, that takes the median of a vector of doubles.
- Part 1. Write the median function using the sort function in the algorithm header.
- Part 2. Write the median function using the partial_sort function in the algorithm header. Do you think this is more efficient? Why? (You can give an intuitive answer without precise mathematical analysis)
- Part 3. Write the median function using the nth_element function header. Do you think this is even more efficient? Why?
- Part 4 - Extra credit: Write a template function that can find the median of a vector of any appropriate type.
  - Although we haven't discussed writing our own template functions yet, looking at the definition of `min` in slide 5 from last week should give you all you need.
  - You can use any of the underlying algorithms from parts 1 to 3 above
- More extra credit. If there are an even number of elements, use the average of the middle 2 element

HW 2.3 (Extra Credit)

- The purpose of this exercise it to show that a well-designed interface is easy to use even if its implementation is very complicated internally.
- Build, compile, and run any program using the optimized_copy function in `optimized_copy.h` (download from chalk). Note that `optimized_copy` is used exactly the same way as `std::copy`, so you should be able to directly adapt any sample code you can find that uses `std::copy`.
- Send something to demonstrate that you've done this successfully (e.g., any files you've written, including C++ files, makefiles, a Visual Studio project files, a transcript of your shell session, etc.)
- Were you able to see a performance improvement over ordinary `std::copy`?

- Do one of the next two problems for full credit
- Do both for extra credit

HW 2.4

- Rewrite the pascal.cpp file in chalk (or your own) to be class-oriented

HW 2.5

- Write a program that takes a sequence of paths, and produces a tree diagram of the files. E.g., let's say that your program is named "tree_print". You'd expect to use this together with find and xargs, e.g.,

  ```
  find . | xargs tree_print
  ```

  should produce a tree oriented print out. For example, suppose I execute the line above in a directory `foo`, which contained file `bar`, and directory `baz` with file `bag`. You'd expect output something like

  ```
  .
     bar
     baz
        bag
  ```

- If you want to (and your system's find command supplies the necessary info), you can put a "/" on the end of directory names
- Your program should be written in a class-oriented fashion.
- "find" is standard on Unix. You can get a copy of the "find" command that runs on Windows at http://gnuwin32.sourceforge.net/packages/findutils.htm. (Make sure you download the prerequisites too).
- You may find `string::find_first_of` and `string::find_last_of` worth looking at. Alternatively, `Boost::Tokenizer` could help.