

C++ Templates

CSsci-3081W: Program Design and Development

Review: Upcasting in C++

- Eraser “is a” Tool.
- Therefore, any pointer to a Tool can point to an Eraser.

```
Eraser *myEraser = new Eraser(...);  
Tool *tool = myEraser;
```

```
// or could do this all in one line:  
Tool *tool1 = new Eraser(...);  
Tool *tool2 = new AirBrush(...);  
Tool *tool3 = new Highlighter(...);
```

Downcasting in C++

- What about the other situation: Can we safely treat any Tool as an Eraser?
- No, a Tool might be an Eraser, but it could also be an AirBrush or a Highlighter, etc.. we don't know for sure.
- In C++ there is a way to turn a Tool* into a Eraser*, but you have to use this with care because this conversion might not always be possible.

Downcasting in C++

- This is how we often do casting in C++, but it's not safe here:

```
Tool *tool = new Eraser(...);
```

```
...
```

```
// This line is not safe -- what if tool doesn't happen to be an eraser?  
Eraser* eraser = (Eraser*)tool;
```

- This is the safe way to do it for downcasting:

```
Tool *tool = new Eraser(...);
```

```
...
```

```
Eraser* eraser = dynamic_cast<Eraser*>(tool);
```

```
if (eraser != NULL) {
```

```
    // It worked.. tool was really an Eraser.
```

```
    ...
```

```
}
```

```
else {
```

```
    // tool is not an Eraser, must be some other type of Tool.
```

```
}
```

A More General Example

```
class A {  
    virtual void f() { }  
};  
  
class B : public A { };  
  
class C { };  
  
void f () {  
    A a;  
    B b;  
  
    // dynamic_cast with pointers:  
    A* ap = &b;  
    B* b1 = dynamic_cast<B*> (&a);    // NULL, because 'a' is not a 'B'  
    B* b2 = dynamic_cast<B*> (ap);    // 'b'  
    C* c = dynamic_cast<C*> (ap);    // NULL, because A is not a C  
  
    // dynamic_cast with references:  
    A& ar = dynamic_cast<A&> (*ap); // Ok.  
    B& br = dynamic_cast<B&> (*ap); // Ok.  
    C& cr = dynamic_cast<C&> (*ap); // throws a std::bad_cast exception  
}
```

Notes on `dynamic_cast`

- Use this sparingly.
- Slight performance hit for real-time type checking.
- But, usually performance isn't the main concern... if you're using `dynamic_cast` all over your code, then that often points to a problem in your code -- you may not be using polymorphism as effectively as you could.

C++ Templates

C++ Templates

- In Java and other languages where all objects derive from some common base “Object”, collections (lists, arrays, sets, etc.) can be defined to hold and return Objects -- this way they will work with any custom objects that you define.
- In C++, there is no “universal superclass”, so we need another mechanism to create generic lists, arrays, sets, etc..
- The solution is to use templates:
 - class templates
 - function templates

What is a template?

- A partial specification of a class or function.
- Each unique instantiation produces a new class or function.
- E.g. if you code declares:
 - `MyArray<int> intArray;`
 - `MyArray<double> doubleArray;`
- Then, there are two classes defined in the running program, even though both come from the same `MyArray` template.
- Because templates are not “real” classes, you almost always include both the declaration and definition code in the `.h` file -- don’t use the usual separation of pieces into `.h` and `.cpp` files.

Template Writing Activity

- Adapted from a very nice reference:
- <http://www.codeproject.com/Articles/257589/An-Idiots-Guide-to-Cplusplus-Templates-Part>

A simple function

```
void PrintTwice(int data) {  
    cout << "Twice is: " << data * 2 << endl;  
}
```

Please write for me a version of printTwice that will work for floats.

A simple function

```
void PrintTwice(int data) {  
    cout << "Twice is: " << data * 2 << endl;  
}
```

```
void PrintTwice(float data) {  
    cout << "Twice is: " << data * 2 << endl;  
}
```

Note, exact same operations, only the type changes.

The compiler has two versions of PrintTwice:

```
void PrintTwice(int data) {...}
```

```
void PrintTwice(float data) {...}
```

First template example:

```
template<typename TYPE>
TYPE Twice(TYPE data) {
    return data * 2;
}
```

```
cout << Twice(10);
cout << Twice(3.14);
```

The compiler would generate two versions:

```
void Twice<int>(int data) {...}
void Twice<float>(float data) {...}
```

- Define a new template function called “Add(n1, n2)”
- Add two objects together, store the result in a local variable within your function.
- Then, return that local variable.
- For reference:

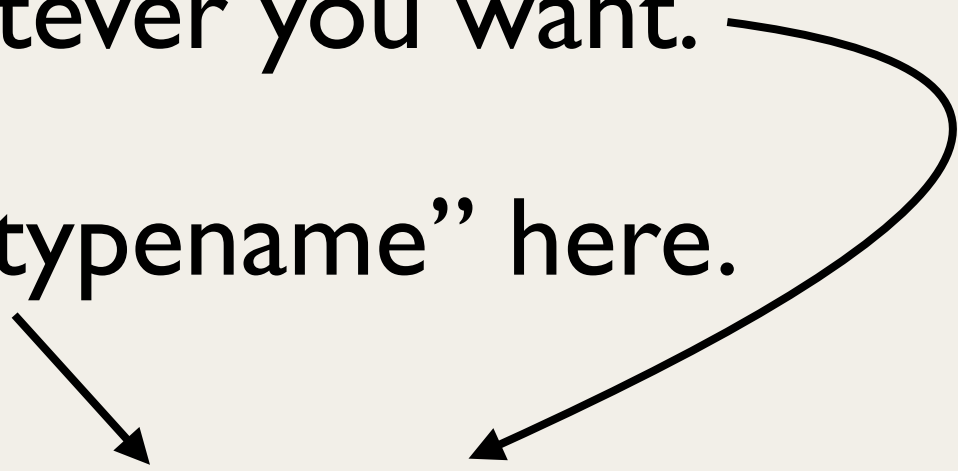
```
template<typename TYPE>  
TYPE Twice(TYPE data) {  
    return data * 2;  
}
```

```
cout << Twice(10);  
cout << Twice(3.14);
```

Add(n1, n2)

Can replace “T” with whatever you want.

Can use either “class” or “typename” here.



```
template<class T>
T Add(T n1, T n2) {
    T result;
    result = n1 + n2;
    return result;
}
```

Add(n1, n2)

```
template<class T1, class T2>
T1 Add(T1 n1, T2 n2) {
    T1 result;
    result = n1 + n2;
    return result;
}
```


Template Class Example: Array

```
template<class T, int size>
class MyArray {
public:
    T& operator[] (int index) {
        assert(index >=0 && index < size);
        return m_array[index];
    }
private:
    T m_array[size];
};
```

```
MyArray<int, 10> myIntArray;
MyArray<double, 500> myDoubleArray;
MyArray<ColorData, 2000> myPixels;
```

Here's a class template challenge for you:

- Remember, some programs store colors as chars: 0-255 and some store as floats: 0.0-1.0.
- Create a ColorData template class that let's you specify what type of storage you would like:
 - `ColorData<char> myColorInChars;`
 - `ColorData<float> myColorInFloats;`

```
template<class T>
class ColorData {
public:
    ColorData(T red, T green, T blue, T alpha) {
        r=red; g=green; b=blue; a=alpha;
    }

    T getRed() { return r; }
    ...

    void setRed(T red) { r = red; }
    ...

private:
    T r,g,b,a;
};
```

```
ColorData<float> data(1.0, 1.0, 1.0, 1.0);
```

```
ColorData<char> data2(255,255,255,255);
```

Templates in the C++ Standard Library

The Standard Template Library (STL)

- Template versions of sequences, sets, maps, queues, and stacks.
- Tuned for efficiency, but depends on the compiler.
- Access to collections (lists, etc.) using iterators.
- Also generic template functions for sorting, searching, etc.
- STL is (partially) included in the C++ standard library, so you can include it via:
 - `#include <deque>`
 - `#include <list>`
 - `#include <set>`
 - `#include <vector>`
 - ...

C++ Standard Library (std) Collection Types

- Ordered Collections (differences are in speed tradeoffs for searching, inserting, erasing):
 - vector: dynamic array that resizes automatically when inserting or erasing an object.
 - list: a doubly-linked list.
 - deque: double-ended queue.
- Unordered Collections
 - set, multiset: a mathematical set
 - map, multimap: an associative array (key-value pairs)
- Adapters (implemented using the other containers):
 - queue: FIFO queue (push, pop, front, back operations)
 - priority_queue: element with highest priority is on top
 - stack: LIFO stack

Containers Common Methods

- `empty()` -- true if the container has no contents.
- `size()` -- the number of elements in the container.
- `max_size()` -- maximum allowed size.
- overloaded `=`, `<`, `>`, `<=`, `>=`, `==`, `!=`
- `swap(c)` -- swaps contents
- `clear()` -- erases contents, calls destructors but does not call delete on pointers.

Using Iterators with Containers

- Iterators are used in conjunction with containers.
- Again, you've already used them in a few instances.
- Iterators are like fancy pointers -- pointers that know the structure of the container.
- Examples:
 - `vector<int>::iterator;`
 - `vector<int>::const_iterator;`

Using Iterators

- Two special iterators that are typically defined are:
 - `begin()` -- returns an iterator to the first element.
 - `end()` -- returns an iterator that's one past the last element.
- Overloaded operators:
 - All iterators support `++`
 - Bidirectional iterators support `--`
 - Random-access iterators support `+` and `+=`
 - You can use `==` to check to see if iterators point to the same thing
 - Like a pointer, you can use `*` to get the object the iterator “points to”

Example with Iterators

- Use `vector<Page>::begin()` and `end()` to get iterators to the start and end of the vector.

```
vector<Page> book;
```

```
// Add a bunch of pages to the book;
```

```
...
```

```
// YOU FILL IN: Use iterators to loop through the book  
// sending each page to cout.
```

Example with Iterators

```
vector<Page> book;
```

```
// Add a bunch of pages to the book;
```

```
...
```

```
vector<Page>::iterator itr;
```

```
for (itr = book.begin(); itr != book.end(); itr++) {
```

```
    // Assume that operator<<() is defined for Page
```

```
    cout << *itr << " ";
```

```
}
```

```
cout << endl;
```

Note: Reverse Iterators

- Iterating backward through a sequence is a bit awkward.
- A `reverse_iterator` can be used.
 - `rbegin()` returns a `reverse_iterator` that is one before the first element.
 - `rend()` returns a `reverse_iterator` that is on the last element.
 - `operator++` runs backwards.

Note: Watch out for Stale Iterators

- Some operations on collections will modify the collection in such a way that will invalidate the iterators, leaving them “stale” -- sort of like a dangling pointer.
- This depends on the particular type of collection that you are using.
- `vector.push_back()` can invalidate all iterators on the vector because the internal storage may need to be reallocated.
- `list.push_back()` does not because the list allocates each element separately.

How to sort a vector<MyClass>

- The standard library can help with some functions/algorithms as well, for example sorting a vector or list, which is great, you don't have to implement the sort routine.
- But, what if your vector holds something like Customers? How should we order customers?
 - By name? By account number? By last transaction date?
...
- The solution is to define < for our classes and then the std::sort routine will know what to do.
 - There are a couple of different ways you could code this: either define the operator< within your class or define a new external function to compute the less than operation.

```

class MyData {
public:
    int m_iData;
    string m_strSomeOtherData;
};

// This is the "binary predicate" function -- used by std::sort()
bool MyDataSortPredicate(const MyData& d1, const MyData& d2) {
    return d1.m_iData < d2.m_iData;
}

int main() {
    // Create and add data to the vector
    vector<MyData> myvector;
    MyData data;
    data.m_iData = 3;
    myvector.push_back(data);

    data.m_iData = 1;
    myvector.push_back(data);

    // Sort the vector using predicate and std::sort
    std::sort(myvector.begin(), myvector.end(), MyDataSortPredicate);

    // Dump the vector to check the result
    for (vector<MyData>::const_iterator citer = myvector.begin();
         citer!=myvector.end(); ++citer) {
        cout << (*citer).m_iData << endl;
    }

    return 1;
}

```

Alternatively...

```
class MyData {  
public:  
    int m_iData;  
    string m_strSomeOtherData;  
    bool operator<(MyData rhs) { return m_iData < rhs.m_iData; }  
};
```

- Then, the call to sort would not need to specify a predicate, just the begin and end iterators:

```
std::sort(myvector.begin(), myvector.end());
```


Other Standard Algorithms

- The header `<algorithm>` defines a collection of functions designed to work on ranges of elements that are passed in by iterators.
- `sort` is one example.
- Other examples:
 - `for_each`: apply function to range
 - `find`: find value in range
 - `fill`: fill range with value
 - `reverse`: reverse range
 - `lower_bound/upper_bound`: return iterator to bound
 - `merge`: merge sorted ranges
 - (and many more: <http://www.cplusplus.com/reference/algorithm/>)

Any Disadvantages of Templates?

- Anecdotally... older versions of the standard template library (STL) were “not so standard” when moving between different architectures and compilers. Thankfully, now, as long as you use the C++ Standard Library, this is not an issue.
- Debugging messages are sometimes really difficult to understand when you use templates because the “name” of each class that you instantiate includes both the template name and the template parameter name.

Smart Pointers

- How many have been burned in the past by “bad” pointers?

Motivation for Smart Pointers

- Memory errors are the worst kind of errors in C++. They crash your program and then can be difficult to debug.
- Java (has other issues, but) doesn't seem to have these same memory issues. Why?
- Java's garbage collection guarantees that a pointer will either point to a valid allocated object or be null.
- If a pointer is either valid or null, then that actually would solve a lot of weird memory errors.
- Can we achieve this in C++?
 - Some people have implemented full-blown garbage collection for C++, but this doesn't quite fit the C++ mindset.
 - A smarter pointer could help though -- in fact, programmers have created many examples of "smart pointers" in C++.

Smart Pointers

- Typically implemented via templates.
- Great reference: <http://ootips.org/yonat/4dev/smart-pointers.html>
- The simplest example is `unique_ptr` from the standard C++ library.

unique_ptr

- Part of unique_ptr's implementation:

```
template <class T> class unique_ptr
{
    T* ptr;
public:
    // Notice, initializes the ptr to NULL
    explicit unique_ptr(T* p = 0) : ptr(p) {}

    // Notice, calls delete automatically
    ~unique_ptr() {delete ptr;}

    // Acts as a "normal" pointer
    T& operator*() {return *ptr;}
    T* operator->() {return ptr;}
    // ...
};
```

unique_ptr

- So, instead of writing:

```
void foo() {  
    MyClass* p = new MyClass;  
    p->DoSomething();  
    delete p;  
}
```

- We can write:

```
void foo() {  
    unique_ptr<MyClass> p(new MyClass);  
    p->DoSomething();  
}
```

Other Smart Pointers

- Pointer that makes a copy of the object it points to when it is copied.
- `shared_ptr`: Shared or reference counted pointers maintain a count of the smart pointers that point to the same object and delete the object when this count becomes zero.
- Different implementations of this idea, some make assumptions about and/or include smarts to handle circular pointers (e.g. parent <--> child):
- http://www.jelovic.com/articles/cpp_without_memory_errors_slides.htm
- http://g3d.sourceforge.net/manual/class_g3_d_1_1_reference_counted_object.html