Eric Roberts
CS 106B

Handout #31
February 9, 2015

Template Structures

Template Structures

Eric Roberts
CS 106B
February 9, 2015

Templates

- One of the most powerful features in C++ is the template facility, which makes it possible to define functions and classes that work for a variety of types.
- The most common form of a template specification is

```
template <typename placeholder>
```

where *placeholder* is an identifier that is used to stand for a specific type when the definition following the **template** specification is compiled.

Templates in Functions

- Templates can be used before a function definition to create a generic collection of functions that can be applied to values of various types.
- The following code, for example, creates a template for the **max** function, which returns the larger of its two arguments:

```
template <typename ValueType>
ValueType max(ValueType v1, ValueType v2) {
   return (v1 > v2) ? v1 : v2;
}
```

- The compiler will generate the code for many different versions of **max**, one for each type that the client uses.
- The function **max** can be used only with types that implement the **>** operator. If you call **max** on some type that doesn't, the compiler will signal an error.

Exercise: Rewrite **sort** as a Template

- Rewrite the following code so that it sorts any type that implements the **<** operator:

```
void sort(int array[], int n) {
   for (int lh = 0; lh < n; lh++) {
      int rh = lh;
      for (int i = lh + 1; i < n; i++) {
         if (array[i] < array[rh]) rh = i;
      }
      swap(array[lh], array[rh]);
   }
}

void swap(int & x, int & y) {
   int tmp = x;
   x = y;
   y = tmp;
}
```

Templates in Class Definitions

- Templates are more commonly used to define generic classes. When they are used in this way, the **template** keyword must appear before the class definition and before each of the implementations of the member functions.
- The most inconvenient aspect of using templates to create generic classes is that the compiler cannot process them correctly unless it has access to both the interface and the implementation at the same time. The effect of this restriction is that the **.h** files for template classes must contain *both* the prototypes and the corresponding code.
- To emphasize the conceptual separation between the interface and the associated implementation, you should make sure to include an appropriate comment before the private section and the implementation warning casual clients away from the details.

A Template Version of the **Stack** Class

- The first step in writing the template version of the **Stack** class is to add the **template** keyword to the interface just before the class definition:

```
template <typename ValueType>
class Stack {
   . . . body of the class . . .
};
```

- Once you have made this change, each instance of the specific type (formerly **char** in the **CharStack** class) must be replaced by the **ValueType** placeholder for the generic type, as in

```
ValueType *elements;
```

or

```
void push(ValueType value);
```

Implementing the Template Class

- The final change necessary to implement the template class is to add **`template`** declarations to every method body, as in the following updated version of the constructor:

```
template <typename ValueType>
Stack<ValueType>::Stack() {
    capacity = INITIAL_CAPACITY;
    count = 0;
    elements = new ValueType[capacity];
}
```

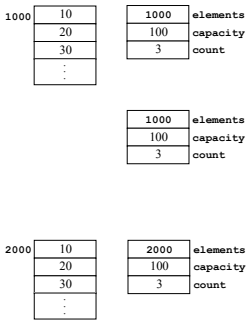
- Because of the restrictions that C++ imposes on template types, the implementations of the methods need to be included as part of the **`stack.h`** header.

Assignment and Copy Constructors

- There is one remaining issue about creating new abstract classes that is extremely important in practice, which is how such objects behave if you copy them using assignment or by passing them by value to parameters in methods.
- The crux of the problem is that copying an abstract data object typically needs to copy the underlying data and not just the fields directly accessible in the object. Unfortunately, the default interpretation in C++ is to copy only the top-level fields, which can lead to serious errors.
- Even though the text includes an extensive discussion of the issues surrounding assignment and copy constructors, we won't hold you responsible for these topics in CS106B. If, however, you are applying for a job that requires you to use C++, you absolutely need to review this material.

Shallow *vs.* Deep Copying

- Suppose that you have a **`Stack<int>`** containing three elements as shown in the diagram to the right.
- A ***shallow copy*** allocates new fields for the object itself and copies the information from the original. Unfortunately, the dynamic array is copied as an address, not the data.
- A ***deep copy*** also copies the contents of the dynamic array and therefore creates two independent structures



Implementing Deep Copy Semantics

- When you are defining a new abstract data type in C++, you typically need to define two methods to ensure that copies are handled correctly:
  - The operator **`operator=`**, which takes care of assignment
  - A ***copy constructor***, which takes care of by-value parameters
- These methods have well-defined signatures and structures, and the easiest thing to do is simply to copy the code on the next slide, adapting it as necessary to account for the specific instance variables that need to be copied in the underlying representation.

Code to Implement Deep Copying

```
/* Code to support deep copying for the Stack class */

template <typename ValueType>
Stack<ValueType>::Stack(const Stack & src) {
   deepCopy(src);
}

template <typename ValueType>
Stack<ValueType> & Stack<ValueType>::operator=(const Stack & src) {
   if (this != &src) {
      if (elements != NULL) delete[] elements;
      deepCopy(src);
   }
   return *this;
}

template <typename ValueType>
void Stack<ValueType>::deepCopy(const Stack & src) {
   count = capacity = src.count;
   elements = (capacity == 0) ? NULL : new ValueType[capacity];
   for (int i = 0; i < count; i++) {
      elements[i] = src.elements[i];
   }
}
```