CSE 165/ENGR 140
Intro to Object Orient Program
Lecture 20 – Templates
Announcement

- Lab #8 open tomorrow
  - Due 4/16 at 11:59PM

- Quiz: 4/16

- Final project out this Friday (in UCMCROPS)
  - Due date: 5/11 (Monday) at 11:59PM
  - Presentation date: 5/12 (Tuesday) at 3:00PM

- Reading assignment:
  - Ch. 1 (Vol. 2)
We have been working with generic objects

- By using the “virtual” method mechanism
- It allows us to develop “object-based container-class hierarchies”:
  - every class has to inherit some generic **Object** class that can be manipulated by containers, etc
  - an Object class object can be downcast to any class type
  - this happens in Java and Smalltalk
What if more than one hierarchy of objects exists?
  ◦ container 1: I need to derive from GraphNodeObject
  ◦ container 2: I need to derive from DoubleLinkObject

One solution: Multiple Inheritance
What if more than one hierarchy of objects exists?
- Example from the book:
  - As classes grow, new classes are needed OShape, OCircle, OSquare, OTriangle, etc
The template solution is different:
  ◦ It tells how a class should be implemented for any (compatible) type

A template class is written for a generic type, and can only be instantiated when the type is given
### Templates

//: C16:Array.cpp

template<class T> // define an array for a generic type T
class Array {
    enum { size = 100 };  
    T A[size]; // T has been used here to create an array of T type

public:
    // operator for array-like access:
    T& operator[](int index) {
        require(index >= 0 && index < size, "Index out of range");
        return A[index];
    }
};

int main() {
    Array<int> ia; // Ok, now the compiler will write and use my Array class for integers
    Array<float> fa; // Now write a new version for float
    for(int i = 0; i < 20; i++) {
        ia[i] = i * i;
        fa[i] = float(i) * 1.414;
    }
    for(int j = 0; j < 20; j++)
        cout << j << " : " << ia[j] << ", " << fa[j] << endl;
}
Templates

- Any type (including your own classes) can be used to instantiate a template.
  - When the compiler “replaces” your type into the template definition, it will check if all operations are supported or not.

- A template class may be written generically to work with many types, or specifically to work with particular classes:
  - If \texttt{T.myMethod()} is called, only types that have a method called \texttt{“myMethod()”} will work, otherwise the compiler will generate an error.
Non-inline Template Methods

//: C16:Array2.cpp
// Non-inline template definition
#include "../require.h"

template<class T>
class Array {
    enum { size = 100 };       
    T A[size];                  
public:
    T& operator[](int index);  
};

template<class T>
T& Array<T>::operator[](int index) {  
    require(index >= 0 && index < size, "Index out of range");
    return A[index];
}

int main() {
    Array<float> fa;
    fa[0] = 1.414;
}
Templates

- All template definitions should appear in the header files only.
  - Anything preceded by `template<...>` means the compiler won’t allocate storage for it at that point, but will instead wait until it’s told to (by a template instantiation).
  - Somewhere in the compiler and linker there’s a mechanism for removing multiple definitions of an identical template.
  - So you’ll always put the entire template declaration and definition in the header file.
Templates can also use constants in the template argument list:

```cpp
template < class T, int size = 100 >
class Array {
    T A[size];

public:
    T& operator[](int index) {
        require(index >= 0 && index < size,"Index out of range");
        return A[index];
    }

    int length() const { return size; }
};
```
You may use "typename" instead of "class"

- The following two cases do the same thing:

```
template < typename T, int size = 100 >
class Array {
    T array[size];
};
```
```
template < class T, int size = 100 >
class Array {
    T array[size];
};
```

"typename" is useful to disambiguate some definitions; for class declarations like above, there is no difference.
Iterators

- An iterator is a class designed to access containers in a generic way.
  - Iterator classes are commonly defined as templates so that they can interface with containers of any type.
  - The standard library has many iterator classes.
  - You can also design iterators for your own classes.
class IntStack {
    enum { ssize = 100 };  
    int stack[ssize];
    int top;
public:
    IntStack() : top(0) {}

    void push(int i) {
        require(top < ssize, "Too many push()es");
        stack[top++] = i;
    }

    int pop() {
        require(top > 0, "Too many pop()s");
        return stack[--top];
    }

    friend class IntStackIter;  // Give full access to the iterator
};
// An iterator is like a "smart" pointer:
class IntStackIter {
    IntStack& s;
    int index;
public:
    IntStackIter(IntStack& is) : s(is), index(0) {}
    int operator++() { // Prefix
        check_error(index < s.top, "iterator moved out of range");
        return s.stack[++index];
    }
    int operator++(int) { // Postfix
        check_error(index < s.top, "iterator moved out of range");
        return s.stack[index++];
    }
};

int main() {
    IntStack is;
    for (int i = 0; i < 20; i++) is.push(fibonacci(i));
    // Traverse with an iterator:
    IntStackIter it(is);
    for (int j = 0; j < 20; j++)
        cout << it++ << endl;
}
It is common to have iterators defined as nested classes, to make clear to what class they work for.

- In the previous example our iterator could be declared as:
  ```cpp
  IntStack::IntStackIter it;
  ```
Iterators

- A lot of functionality can be added to iterators, for example to achieve instructions such as this one:

```cpp
while (start != end)
    cout << start++ << endl;
```

- In the Standard C++ Library, the start and end iterators are produced by the container member functions `begin()` and `end()`. 
Why Iterators?

- It may allow you to write algorithms independent of the container chosen to hold your data:
  - a print() method can be designed to receive a generic iterator that can be used to traverse and print the elements of different containers.
  - a sort() method can work with arrays and linked lists if a common iterator with suitable methods exist.

- It allows, some times, to design more efficient, generic and safer classes.
  - Sometimes iterators need to save internal states of a traversal (so that multiple traversals do not conflict)
    - If each iterator keeps track of its traversal state there are no conflicts, any number of traversals is supported, and the container class will hold less data.
The compiler will try to figure out automatically the type of function templates:

```cpp
template<typename X>
X Max ( X a, X b )
{ return a>b? a:b; }

int main() {
  int i = Max ( 3, 4 );
  float f = Max ( 3.1f, 4.1f );
  double d = Max ( 3.1, 4.1 );
  double x = Max ( 1, 4.1 ); // Error: template parameter is ambiguous
  return 0;
}
```
The code example below automatically compiles; but if “explicit” is written before the constructor, it won’t:

class X
{ public:
   X (int i) { m=i; }
   int get () { return m; }
private:
   int m;
};

void f ( X x ) { int i = x.get(); }

int main ()
{
   f (42);
}
explicit Keyword

- This version will generate an error:

```cpp
class X
{
public:
    explicit X (int i) { m=i; }
    int get () { return m; }
private:
    int m;
};

void f ( X x ) { int i = x.get(); }

int main ()
{
    f (42); // Error: no suitable constructor exists
}
```
A class template may have variations for specific types of instantiation.

For that, use the template<> declaration:

```cpp
// A min() template
template<class T>
const T& min(const T& a, const T& b) {
    return (a < b) ? a : b;
}

// An explicit specialization of the min template
template<>
const char* const& min<const char*>(const char* const& a, const char* const& b) {
    return (strcmp(a, b) < 0) ? a : b;
}
```
One Advanced Example

- Template Programming
  - It is possible to use templates to perform computations for us at compilation time, not run-time!
  
  - The following example will “unroll” many template definitions and compute the result of a Fibonacci sum at compilation time.

  - Example shown in next slide

  (more examples in the book volume 2)
One Advanced Example

template<int n>
struct Fib {
    enum {
        val = Fib<n-1>::val + Fib<n-2>::val
    };
};

template<>
struct Fib<1> {
    enum {
        val=1
    };
};

template<>
struct Fib<0> {
    enum {
        val=0
    };
};

int main() {
    cout << Fib<5>::val << endl; // 6
    cout << Fib<20>::val << endl; // 6765
}
Exercise:
   - Re-write the book’s Stash class using templates so that you can achieve a dynamic array for any object type (similarly to std::vector<>)

Templates