

CS 32

Lecture 5: Templates



Vectors

- Sort of like what you remember from Physics...
- ...but sort of not
- Can have many “components”
 - Usually called entries
- Like Python lists

Array vs. Vector

- What's the difference?
 - In some languages, Array means what Vector means here
- Array is C data type
- Vector is C++ class (really template)

Static Array

- `int x[10];`
 - Locally allocated (auto/stack)
 - Starts out filled with 0 (or junk)
 - Statically laid out
 - Fixed size

Dynamic Vector

- `vector<int> v;`
 - Starts out empty
 - New entries allocated dynamically
 - Length can increase
 - Made to look like it's an array

Accessing Vectors

- `v.push_back(3);`
 - Add 3 to end of vector; $O(1)$
- `int i = v[5];`
 - `v` is an rvalue; $O(1)$
- `v[7] = 8;`
 - `v` is an lvalue; $O(1)$

In-between Way

- Using pointers
- `int* x = new int[10];`
- `int* x = (int *)malloc(10)`
`// old-school`
- x can be passed outside of scope
- Storage in “free store” (heap)
- You must delete it

Those Brackets

- So what was up with the angle brackets?
 - `vector<int> v;`
 - Or were they less-than/gt-than>?
- Used widely for holding parameters
 - Specifically type parameters

First Example

```
template< class ForwardIt, class T >  
bool binary_search( ForwardIt first, ForwardIt last, const T& value )
```

- We already saw type params in use
- Creating a template is abstraction
 - Variable part is factored-out
- Using a template is specialization
 - A type is plugged in to dummy variable T

DRY

- Factoring is often motivated by this
- You write identical (or almost) code in several places
- Factor out the commonalities into a parameter

Value Params

```
int a = 2*3 + 2*7;  
int b = 2*5 + 2*6;  
int c = 2*11 + 2*77;
```

// Why am I redoing this?

```
int doubleSum(int x, int y) {  
    return 2*x + 2*y;  
}  
int a = doubleSum(3, 7);  
int b = doubleSum(5, 6);  
int c = doubleSum(11, 77);
```

Type Params

```
template<typename T>
void myGreatFunc(T& param1) {
    // Do a lot of stuff that doesn't depend
    // on the actual type T
}
```

- It's just a convention to use T
- Book warning (p. 931) about not writing template declarations
- Remember declaration vs. definition?

Example

- Swapping values
- Classic example
- Always requires a temp variable
- So always vulnerable to thread-switching
- OSes provide “atomic” swap call

Generic Swap

```
template<typename T>
void swap(T& v1, T&v2) {
    T temp = v1;
    v1 = v2;
    v2 = temp;
}
```

- Type of swapped items is abstract
- You wrote this code a few times, said “[bad words] I’m factoring this.”

Under the Hood

- What happens when you use the template?
- The compiler uses the template to create real function with the type filled in
- So it is doing prototype-based programming

Using It

- You write

```
int i = 3; int j = 4;  
swap(i,j); // Uh-oh, time to create an integer swap.
```

- The compiler creates

```
void swap(int& v1, int&v2) {  
    int temp = v1;  
    v1 = v2;  
    v2 = temp;  
}
```

- Type parameter `int` has been plugged in

Build On That

```
template<typename T>  
void swap(T& v1, T&v2) { // etc. }
```

```
template<typename BaseType>  
int index_of_smallest(const BaseType a[], ... ) {  
    // As before in Ch. 7  
}
```

```
template<typename BaseType>  
void sort(BaseType a[], ... ) {  
    // Uses index_of_smallest()  
}
```

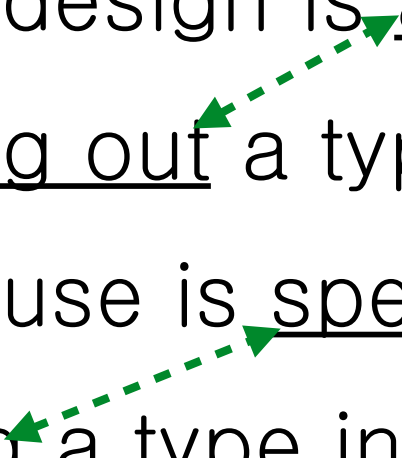
- Get a generic sorting function

Classy Templates

- Can pull the same trick with classes as with functions
- Looks and works pretty much similarly

```
template<typename T>  
class MyGreatClass {...}
```

Say It Again

- Template design is abstraction
 - Factoring out a type param
 - Template use is specialization
 - Plugging a type into the param
- 

Fundamental The

Synonyms	Antonyms	
	(Harder)	(Easier)
	Factor out	Plug in
	Parameterize	Hard-code
	Abstract	Specific
	Generalize	Specialize
	Integrate	Differentiate
	Synthesize	Analyze
	Innovate	Imitate
	Predict	Review

STL

- Not Saint Louis airport code
- Not Standard Telegraph Level
- Standard Template Library
 - Added in the 90s

STL Highlights

- vector
- stack (LIFO)
- queue (FIFO)
- set
- map
- iterators

Iterators

- More abstract kind of pointer
- Used for same purposes
- Made to look like a pointer
 - I.e. has ++, ==, *, etc. defined
- Point into “container classes”
 - vector, queue, ...
 - Container gives you the iterator

Out With The Old...

- Old-school

```
int* lots = // Lots of ints  
for (int* p = lots; p < whatever; p++) {...}
```

- New-school

```
// c is an STL container  
for (p = c.begin(); p != c.end(); p++) {...}
```


Tastes Like Chicken

- Define the iterator

```
// c is an STL container  
vector<char>::iterator p = c.begin();
```

- All these are the same
 - `c[2]`
 - `p[2]`
 - `*(p + 2)`

Im/mutable

- The most important distinction
- Const iterators don't let you change the item referred to
- `vector<int>::const_iterator p = ...`

Containers

- You feel entitled to have them after using a scripting language
- `vector`
 - Like a dynamic array
- `list`
 - Kinda the same
- What's the difference?

list

- In C++, has specifically defined time complexities
 - Essentially this means it's a doubly-linked list
 - Inserting/removing much faster than with vector

slist

- Singly-linked list
 - More efficient insertion/deletion
 - Uses less memory than list
 - Can only iterate forward

And the Rest

- queue (LIFO)
 - For things waiting to be dealt with
- stack (FIFO)
 - For nested things
- set
 - For unordered things

The Best

- `map`
 - also called that in Java, Go
 - `Dict` in Python, Swift, Obj-C, C#
 - `Hash` in Perl, Ruby
 - `Objects` in JavaScript

Motivation

- Using integers to index arrays gets boring
- Want more general mapping into container
- Especially want `container["abc"]` to work

What's Allowed In?

- Hash parameter (called “key”) must be “hashable”
- I.e. be able to create from it a number in a particular range
- Is “almost” unique

Looks the Same

- In Swift

```
public struct Dictionary<Key : Hashable, Value> : Collection {
```

```
public protocol Hashable : Equatable {  
    public var hashCode: Int { get }
```

```
public protocol Equatable {  
    public static func ==(lhs: Self, rhs: Self) -> Bool
```

Read!

- Problem Solving 10.2
 - Which is Scoping and Encapsulation