
Container Types

Reid Watson
(rawatson@stanford.edu)

Administrivia

- Happy Halloween!

What We Covered Tuesday

- Creating our own types
 - Defined a "point" class representing a two dimensional point
 - Gave point a constructor
 - Gave point member functions
 - Gave point member variables
 - Gave point static member variables
 - Gave point private member functions
-

Goals for Today

- Design our own vector class
 - Make it work for any type
 - Make it work (almost) as well as a standard library vector
-

Designing Vector

- You've already seen the implementation of a vector in lecture (CS106B) or when you wrote PQueue
 - We're going to look at the interaction between a vector, iterators and templates
 - When we finish, you should be able to understand a lot more of Stanford's **Vector.h** header
-

Reference Interlude

- Before that, we need to learn more fun things you can do with references
 - References in function parameters
 - Local reference variables
 - References returned from functions
-

Reference Interlude

- Before we start talking about vectors I want to review references:
 - Specifically I want to talk about three uses of references:
 - References in function parameters
 - Local reference variables
 - References returned from functions
-

Reference Interlude

- You've already seen functions with reference parameters:

```
void f(int& x) {  
    x += 1;  
}  
  
int main() {  
    int x = 1;  
    f(x);  
    cout << x << endl; // prints 2  
}
```

Reference Interlude

- Before we start talking about vectors I want to review references:
 - Specifically I want to talk about three uses of references:
 - References in function parameters
 - **Local reference variables**
 - References returned from functions
-

Reference Interlude

- You can also create references like any other variable in a function:

```
int main() {  
    int x = 1;  
    int& y = x;  
    y = 2;  
    cout << x << endl; // prints 2  
}
```

Reference Interlude

- Reference variables inside a function can be useful for saving a result:

```
// This function takes a long time to run  
int findImportantIndex();
```

```
cout << elems[findImportantIndex()] << endl;  
elems[findImportantIndex()].doThings();  
elems[findImportantIndex()].add(2);  
// Why is this bad?
```

Reference Interlude

- We could avoid the function call by saving the computed index

```
// This function takes a long time to run  
int importantIndex = findImportantIndex();  
  
cout << elems[importantIndex] << endl;  
elems[importantIndex].doThings();  
elems[importantIndex].add(2);
```

Reference Interlude

- Even better, we could save the element itself
- This is faster and more concise

```
// This function takes a long time to run  
Foo& important = elems[findImportantIndex()];
```

```
cout << important << endl;  
important.doThings();  
important.add(2);
```

Reference Interlude

- Before we start talking about vectors I want to review references:
 - Specifically I want to talk about three uses of references:
 - References in function parameters
 - Local reference variables
 - **References returned from functions**
-

Reference Interlude

- Functions can also return references:

```
int global = 1;
int& getGlobal() {
    return global;
}
```

```
int main() {
    getGlobal() += 1;
    cout << global << endl; // prints 2
}
```

Reference Interlude

- Here's a case where this is more useful:

```
struct Person {  
    string nameVar;  
    string name() {return nameVar;}  
};  
  
int main() {  
    Person p = {"Joe"};  
    p.name() = "Bob"; // what does this line do?  
    cout << p.name() << endl;  
}
```

Reference Interlude

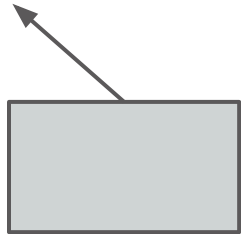
- Let's say we add the reference:

```
struct Person {  
    string nameVar;  
    string& name() {return nameVar;}  
};  
  
int main() {  
    Person p = {"Joe"};  
    p.name() = "Bob"; // what does this line do?  
    cout << p.name() << endl;  
}
```

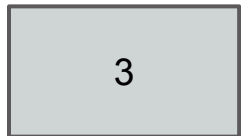
Designing Vector

Let's now start looking at how to build a vector

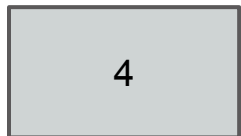
Designing Vector



Data Pointer (**e**lems)

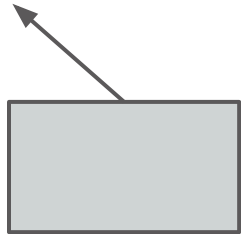


Logical Size (**l**ogicalSize)

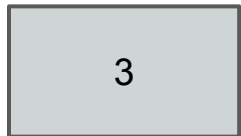


Allocated Size (**a**llocatedSize)

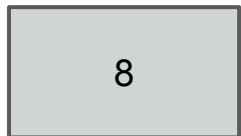
Designing Vector



Data Pointer (**e**lems)

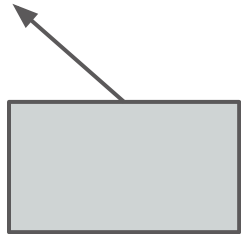


Logical Size (**l**ogicalSize)



Allocated Size (**a**llocatedSize)

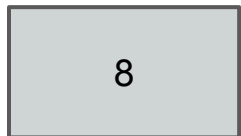
Designing Vector



Data Pointer (**e**lems)



Logical Size (**l**ogicalSize)



Allocated Size (**a**llocatedSize)

Designing Vector

- Constructor and destructor
 - Get logical size
 - Define iterator types and an accessor
 - Get and modify allocated size
 - Write methods to insert values into our vector
-

Designing Vector

Let's start by looking at the basic constructor /
destructor

Designing Vector

Now let's try writing functions to get the logical size of a vector and check if it's empty

Designing Vector

Now let's try and define our iterator type and create begin and end methods.

While we're at it we can define a method to access an element of our vector

Designing Vector

Next step: we need to be able to get and change the allocated size of our vector

Designing Vector

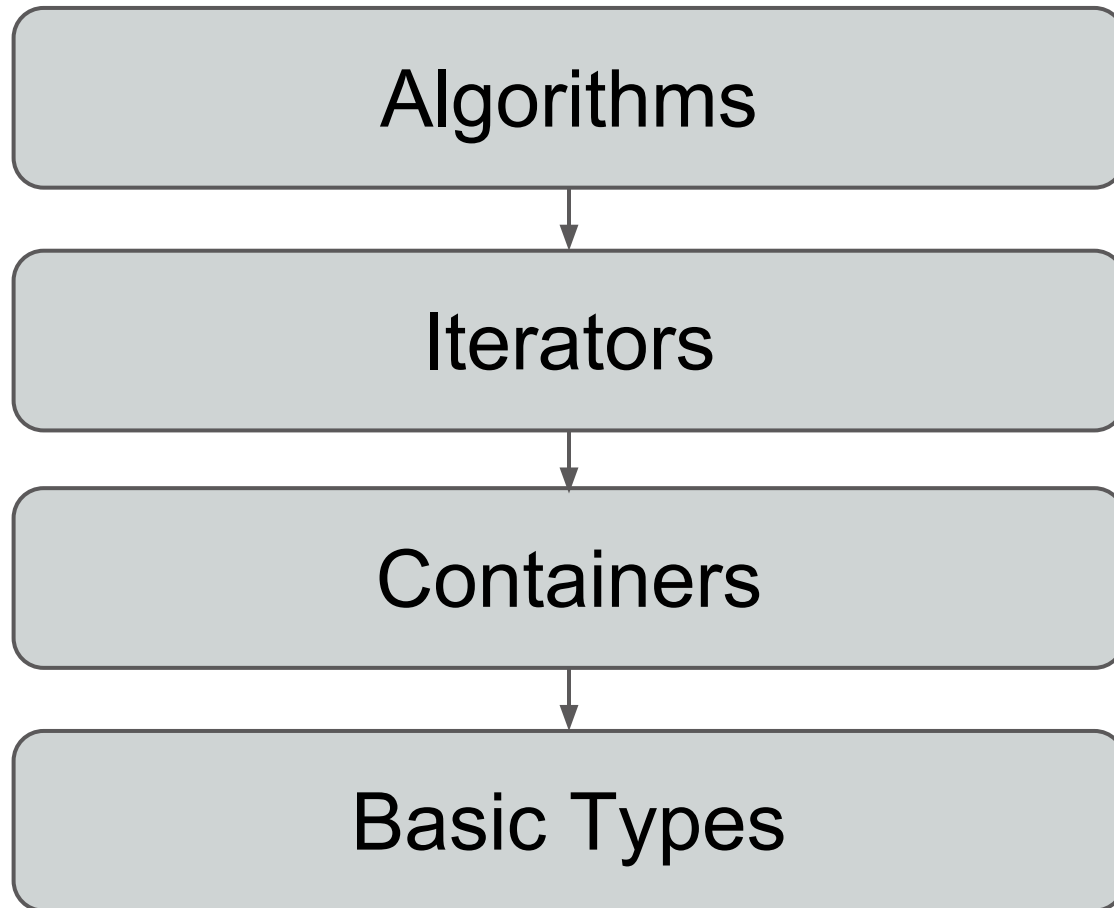
Finally, let's write methods to insert values into our vector

Review

We're now ready to start talking about class templates.

Let's review templates in general first.

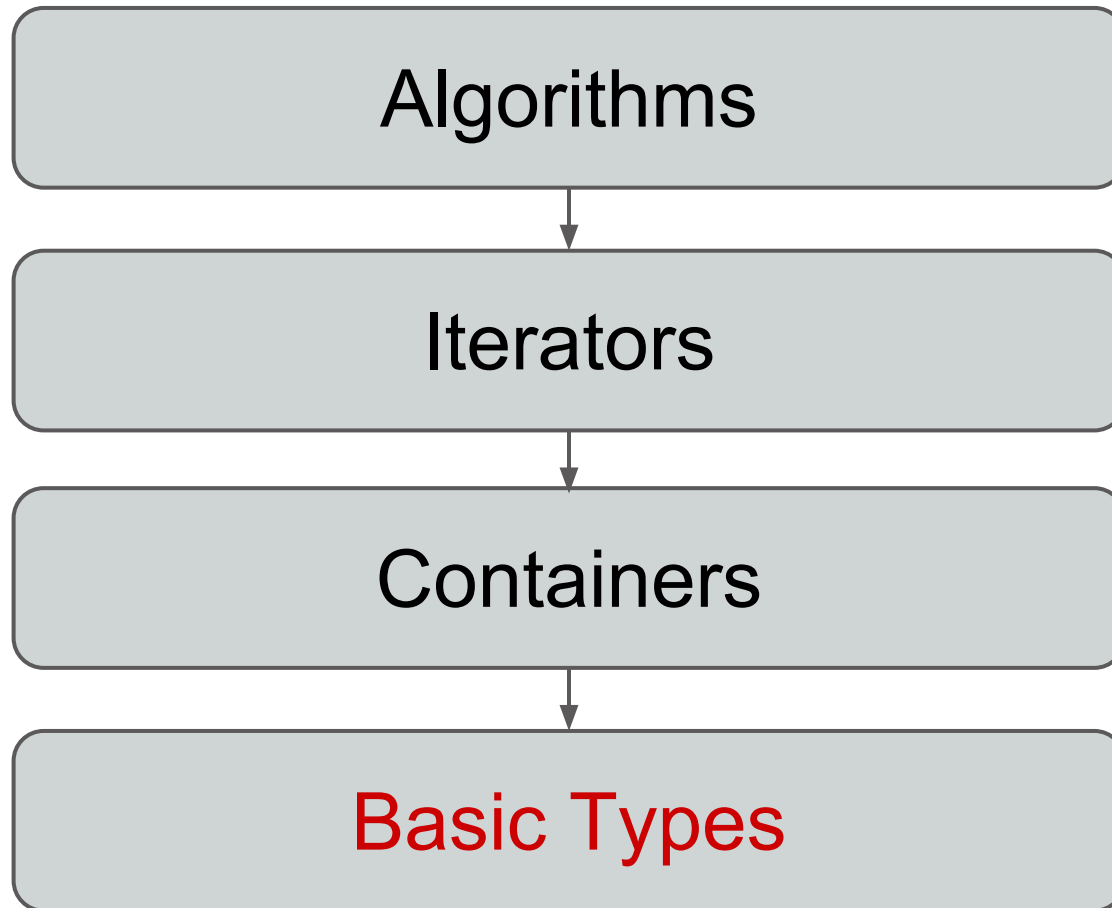
Review



Review

We all understand the basic types (**int**, **char**, **bool**, etc).

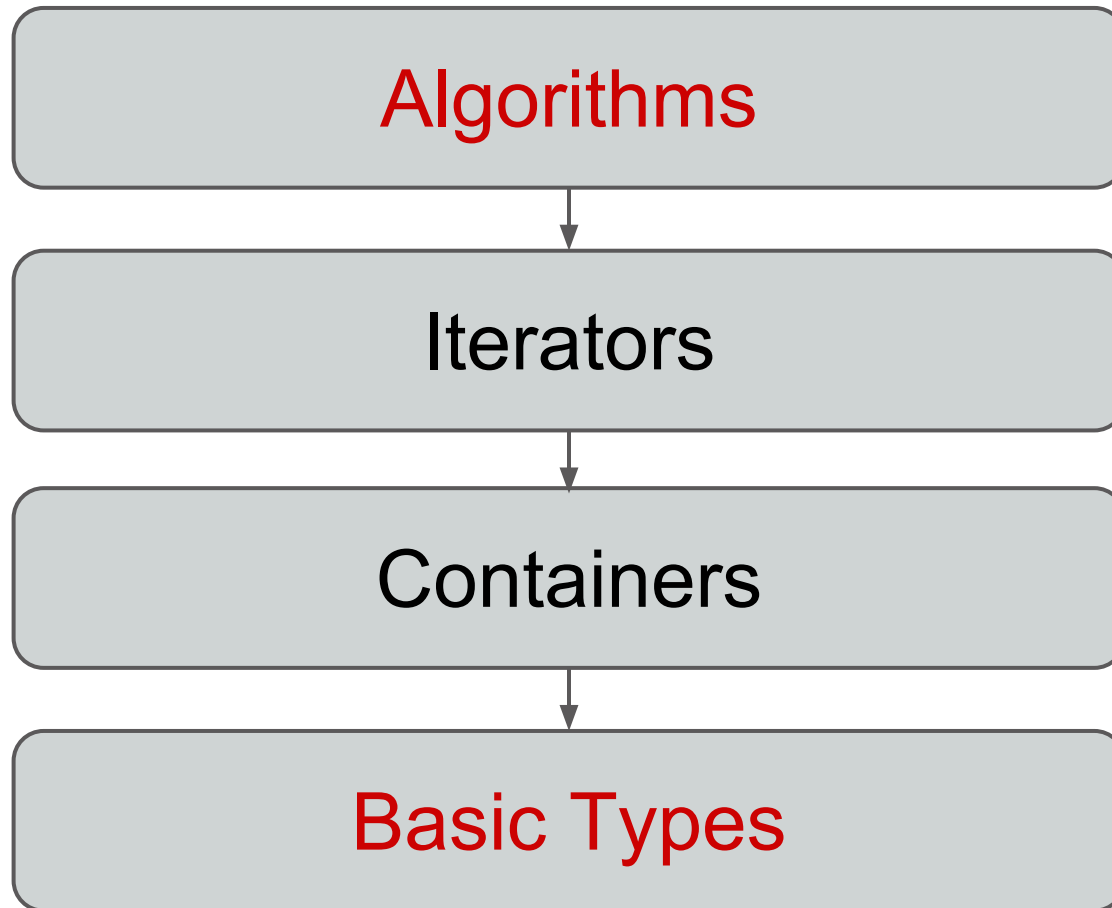
Review



Review

We've also learned how template functions work, so we should be ready to implement our own algorithms.

Review



Review

There are still some missing pieces.

We haven't really figured out how containers abstract away basic types.

That is, how is it that we can use the same code for both vectors of **ints** and vectors of **chars**?

Function Template

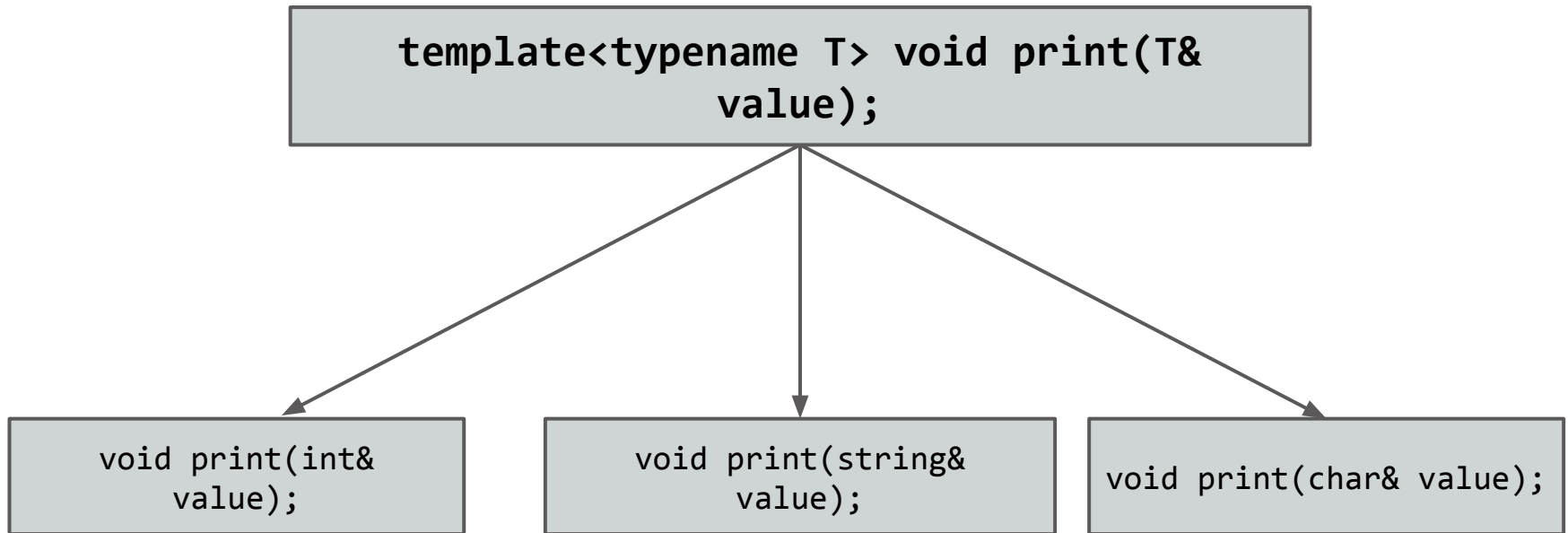
We've already talked about how to create functions which work for any type (function templates)

Function Templates

```
template<typename T>
void print(T& value) {
    cout << value << endl;
}

print(1);
print("Hello!");
print('a');
```

Function Templates



We **instantiate** our **function template** for whatever types we need to use. In doing so, we verify that it's a reasonable instantiation.

Class Templates

We can apply the same principle to create a class which can store any type of variable.

We'll define a **class template** and **instantiate** it for any types that we need to create.

Let's see why this is useful.

Class Templates

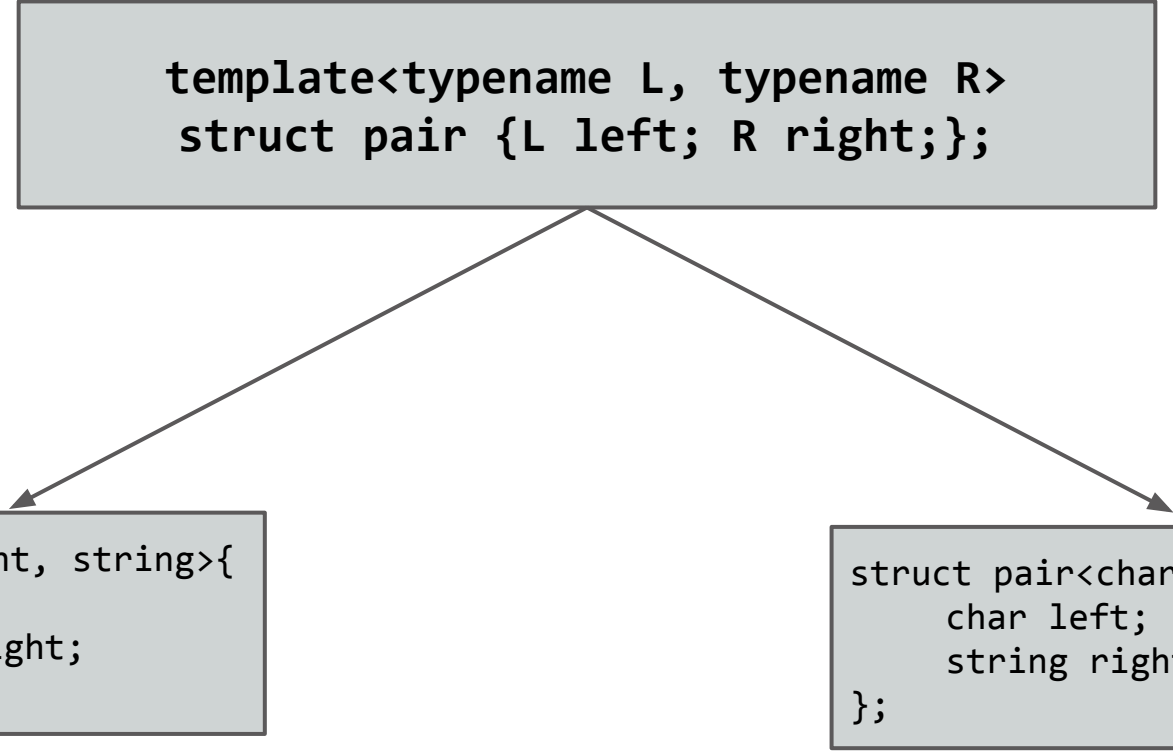
```
// The boring way...  
struct intAndString {  
    int first;  
    string second;  
};  
struct charAndString {  
    char first;  
    string second;  
};
```

Class Templates

```
template <typename L, typename R>
struct pair {
    L left;
    R right;
};
pair<int, string> x = {1, "2"};
pair<char, string> y = {'a', "2"};
```

Class Templates

```
template<typename L, typename R>  
struct pair {L left; R right;};
```



```
graph TD; A["template<typename L, typename R>  
struct pair {L left; R right;};"] --> B["struct pair<int, string>{  
    int left;  
    string right;  
};"]; A --> C["struct pair<char, string>{  
    char left;  
    string right;  
};"];
```

```
struct pair<int, string>{  
    int left;  
    string right;  
};
```

```
struct pair<char, string>{  
    char left;  
    string right;  
};
```

We **instantiate** our **class template** for whatever types we need to use.

Class Templates

Let's take a look at this in code

Class Templates: Gory Details

- When we define a class template, we **only** use a .h file, and **do not** define member functions in a .cpp file
 - This is contrary to how things are done normally
 - Member functions are defined differently
 - There's a bit of weird syntax for accessing **nested types**
-

Class Templates: Gory Details

- When we define a class template, we **only** use a .h file, and **do not** define member functions in a .cpp file
 - This is contrary to how things are done normally
 - Member functions are defined differently
 - There's a bit of weird syntax for accessing **nested types**
-

Class Templates: Gory Details

- When we define a class template, we **only** use a .h file, and **do not** define member functions in a .cpp file
 - This is contrary to how things are done normally
 - **Member functions are defined differently**
 - There's a bit of weird syntax for accessing **nested types**
-

Class Templates: Gory Details

- When we define a class template, we **only** use a .h file, and **do not** define member functions in a .cpp file
 - This is contrary to how things are done normally
 - Member functions are defined differently
 - There's a bit of weird syntax for accessing **nested types**
-