Templatised Classes

Ali Malik
malikali@stanford.edu
Game Plan

References
Designing ADTs
Templatised Classes
Some Subtleties
Announcements
Designing Vector

You’ve already seen the implementation of Vector in CS106B:

- Allocate initial capacity
- When full, allocate double the memory and copy over old elements.

But...
Designing Vector

The VectorInt Class: Implementation

- In order to demonstrate how useful (and necessary) dynamic memory is, let's implement a Vector that has the following properties:
  - It can hold ints (unfortunately, it is beyond the scope of this class to create a Vector that can hold any type)
  - It has useful Vector functions: add(), insert(), get(), remove(), isEmpty(), size(), << overload
  - We can add as many elements as we would like
  - It cleans up its own memory
Designing Vector

The VectorInt Class: Implementation

- In order to demonstrate how useful (and necessary) dynamic memory is, let's implement a Vector that has the following properties:
  - It can hold `ints` (unfortunately, it is beyond the scope of this class to create a Vector that can hold *any* type)
  - It has useful Vector functions: `add()`, `insert()`, `get()`, `remove()`, `isEmpty()`, `size()`, `<< overload`
  - We can add as many elements as we would like
  - It cleans up its own memory
Designing Vector

We will write a full fledged Vector class:

- Templatised
- Const correct
- Provides Iterators
References

Another name for an already **existing** object.

```cpp
int x = 15;
int &refToX = x;
refToX = 3; // refToX is a synonym for x
cout << x << endl; // prints 3
```
References

Can be used as local variables:

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

cout << elems[findIndex()] << endl;
elems[findIndex()].doThings();
elems[findIndex()].add(2);
// excessive calls to slow function
```
References

Can be used as local variables:

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

cout << elems[findIndex()] << endl;
elems[findIndex()].doThings();
elems[findIndex()].add(2);
// excessive calls to slow function
```
Can be used as local variables:

```cpp
// This function takes a long time to run
int indx = findIndex();

cout << elems[findIndex()] << endl;
elems[findIndex()].doThings();
elems[findIndex()].add(2);
// excessive calls to slow function
```
// This function takes a long time to run
int indx = findIndex();

cout << elems[findIndex()] << endl;
elems[findIndex()].doThings();
elems[findIndex()].add(2);
// excessive calls to slow function
References

Can be used as local variables:

```cpp
// This function takes a long time to run
int indx = findIndex();

cout << elems[indx] << endl;
elems[indx].doThings();
elems[indx].add(2);
// no redundant function calls
```
This function takes a long time to run

```cpp
int indx = findIndex();

cout << elems[indx] << endl;
elems[indx].doThings();
elems[indx].add(2);
// no redundant function calls
```

Better, but not the best.
References

Can be used as local variables:

```cpp
// This function takes a long time to run
int indx = findIndex();

cout << elems[indx] << endl;
elems[indx].doThings();
elems[indx].add(2);
// no redundant function calls
```
// This function takes a long time to run
Foo& curr = elems[findIndex()];

cout << curr << endl;
curr.doThings();
curr.add(2);
// no redundant accessing of elems vector

References

Can be used as local variables:

We can have a reference to the element we want to modify
References

Can be returned by functions

```c
int global = 1;

int& getGlobal() {
    return global;
}

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

Can be returned by functions

```c
// REALLY BAD
int& getGlobal() {
    int x = 5;
    return x;
}

int main() {
    getGlobal() += 2;       // undefined behaviour
    cout << global << endl;
}
```
References

Can be returned by functions

```cpp
// REALLY BAD
int& getGlobal() {
    int x = 5;
    return x;
}

int main() {
    getGlobal() += 2;  // undefined behaviour
    cout << global << endl;
}
```
How do References Work?

Not specified by the standard.

Usually implemented by the compiler as pointers that get automatically dereferenced:

```c
int x = 3;
int& refToX = x;
++refToX;
```

```c
int x = 3;
int* refToX = &x;
++(*refToX);
```
Designing MyVector
Designing Vector

- Define iterator and size types
- Default Constructor
- Fill Constructor
- Destructor
- operator[] and at() methods
- Getters for size, empty, begin iterator, end iterator
- Insert and push_back methods
Implementing StrVector

Let’s implement a quick string vector:

StrVector.pro
Templated Classes
Recap - Function Templates

We can give that blueprint to the compiler in the form of a template function by telling it what specific parts need to get replaced.

Just before the function we specify a template parameter.

```c
int min(int a, int b) {
    return (a < b) ? a : b;
}
```
Recap - Function Templates

We can give that blueprint to the compiler in the form of a template function by telling it what specific parts need to get replaced.

Just before the function we specify a template parameter.

```c
int min(int a, int b) {
    return (a < b) ? a : b;
}
```

Let’s make this general.
Recap - Function Templates

We can give that blueprint to the compiler in the form of a template function by telling it what specific parts need to get replaced.

Just before the function we specify a template parameter.

```c++
T min(T a, T b) {
    return (a < b) ? a : b;
}
```

Some generic type T.
Recap - Function Templates

We can give that blueprint to the compiler in the form of a **template function** by telling it what specific parts need to get replaced.

Just before the function we specify a **template parameter**.

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

Tell compiler `T` is a generic type.
Recap - Function Templates

We can give that blueprint to the compiler in the form of a template function by telling it what specific parts need to get replaced.

Just before the function we specify a template parameter.

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

Tell compiler T is a generic type.

It will replace the parameter for us!
Class Templates

The idea with class templates is the same.
A few more annoying nuances to watch out for.

class StrVector {

public:
    void push_back(const std::string& elem) ;
    // rest of implementation

}
Class Templates

The idea with class templates is the same.
A few more annoying nuances to watch out for.

class StrVector {

public:
    void push_back(const std::string& elem) ;
    // rest of implementation
}

The idea with class templates is the same.

A few more annoying nuances to watch out for.

class StrVector {

public:
    void push_back(const ValueType& elem) ;
    // rest of implementation
}
Class Templates

The idea with class templates is the same.

A few more annoying nuances to watch out for.

template <typename ValueType>
class StrVector {

public:
    void push_back(const ValueType& elem);
    // rest of implementation
}
Class Templates

Let’s modify our class to be templatised:

MyVector.pro
Templatised Classes
The Gory Details
Class Templates - Details

When we define a class template, we **only** use a .h file, and **do not** define member functions in a .cpp file.

Member functions are defined differently.

There's a bit of weird syntax for accessing nested types.
Class Templates - Details

Must announce that every method is templated

template <typename ValueType>
class Vector {

public:
    void push_back(const ValueType& elem);
    // rest of implementation

}  

void Vector::push_back(const ValueType& val) {

}
Class Templates - Details

Must announce that every method is templated

```cpp
template<typename ValueType>
class Vector {

public:
    void push_back(const ValueType& elem);
    // rest of implementation

} // Compiler error

void Vector::push_back(const ValueType& val) {

}
```
Class Templates - Details

Must announce that every method is templated

template <typename ValueType>
class Vector {

public:
    void push_back(const ValueType& elem);
    // rest of implementation
}

void Vector::push_back(const ValueType& val) {

}
Class Templates - Details

Must announce that every method is templated

template<typename ValueType>
class Vector {

public:
    void push_back(const ValueType& elem);
    // rest of implementation
}

void Vector::push_back(const ValueType& val) {

}
Class Templates - Details

Must announce that every method is templated

template <typename ValueType>
class Vector {

public:
    void push_back(const ValueType& elem);
    // rest of implementation
}

void Vector<ValueType>::push_back(const ValueType& val) {

}
Class Templates - Details

Must announce that every method is templated

template <typename ValueType>
class Vector {

public:
    void push_back(const ValueType& elem);
    // rest of implementation
}

void Vector<
ValueType>::push_back(const ValueType& val) {

}
Class Templates - Details

Must announce that every method is templated

template <typename ValueType>
class Vector {

public:
    void push_back(const ValueType& elem);
    // rest of implementation
}

void Vector<ValueType>::push_back(const ValueType& val) {
}
Class Templates - Details

Must announce that every method is templated

template <typename ValueType>
class Vector {

public:
    void push_back(const ValueType& elem);
    // rest of implementation
}

template <typename ValueType>
void Vector<ValueType>::push_back(const ValueType& val) {
}

Class Templates - Details

Must announce that every method is templated

template <typename ValueType>
class Vector {

public:
    void push_back(const ValueType& elem);
    // rest of implementation
}

template <typename ValueType>
void Vector<ValueType>::push_back(const ValueType& val) {

}
Class Templates - Details II

Must use `typename` keyword for nested types:

```cpp
template <typename ValueType>
Vector<ValueType>::iterator Vector<ValueType>::push_back(const ValueType& val) {
}
```
Must use `typename` keyword for nested types:

```cpp
template <typename ValueType>
Vector<ValueType>::iterator Vector<ValueType>::push_back(const ValueType& val) {
    return this->begin();
}
```

Error:

```
missing 'typename' prior to dependent type name 'Vector<ValueType>::iterator'
```

```
/Users/alimalik/Desktop/Programs/C++/StringVector/strvector.h
```
Class Templates - Details II

Must use `typename` keyword for nested types:

```cpp
template <typename ValueType>
Vector<ValueType>::iterator Vector<ValueType>::push_back(const ValueType& val) {
}
```
Class Templates - Details II

Must use `typename` keyword for nested types:

```cpp
template <typename ValueType>
typename Vector<ValueType>::::iterator
    Vector<ValueType>::::push_back(const ValueType& val) {
}
```
Next Time
Constructors