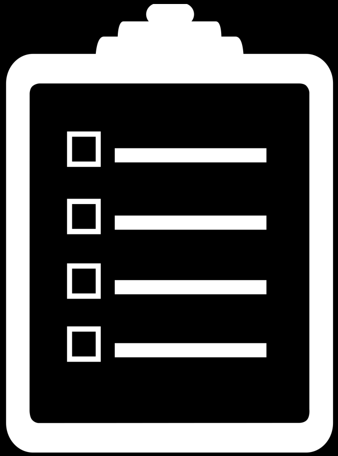
Templatised Classes

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:

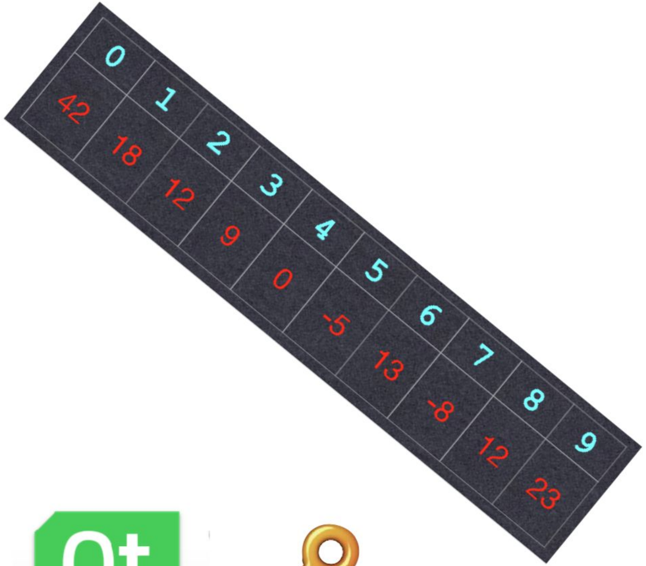
- Allocate initial capacity
- When full, allocate double the memory and copy over old elements.

But...

Designing Vector

The VectorInt Class: Implementation

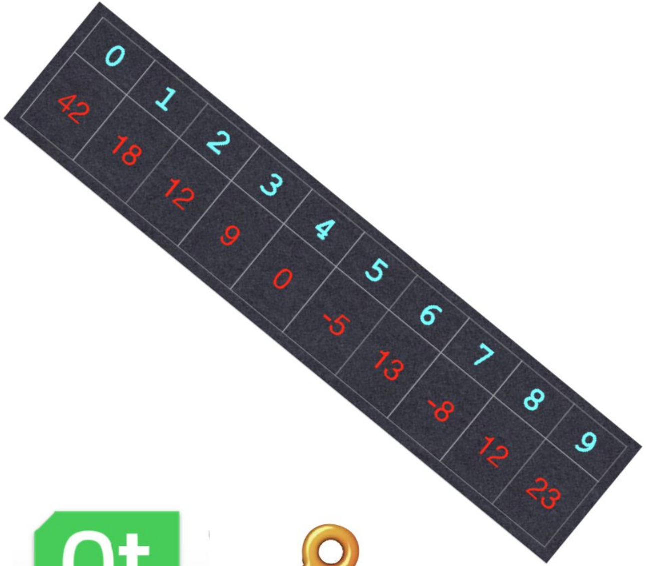
- In order to demonstrate how useful (and necessary) dynamic memory is, let's implement a Vector that has the following properties:
  - It can hold **`int`**s (unfortunately, it is beyond the scope of this class to create a Vector that can hold *any* type)
  - It has useful Vector functions: **`add()`**, **`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 **`int`**s (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

References

Another name for an already existing object.

```
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:

```
// 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:

```
// 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:

```
// 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:

```
// 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:

```
// 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
```

References

Can be used as local variables:

```
// 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
```

Better, but not the best.

References

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

Can be used as local variables:

```
// 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
```

References

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

Can be used as local variables:

```
// 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 returned by functions

```
int global = 1;

int& getGlobal() {
    return global;
}

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

References

Can be returned by functions

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

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

References

Can be returned by functions

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

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

Return by reference mostly used with dynamic memory allocation or stream operators

How do References Work?

Not specified by the standard.

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

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

→

```
int x = 3;
int* refToX = &x;
++(*refToX);
```

Const Interface

Classes have const and non-const interfaces

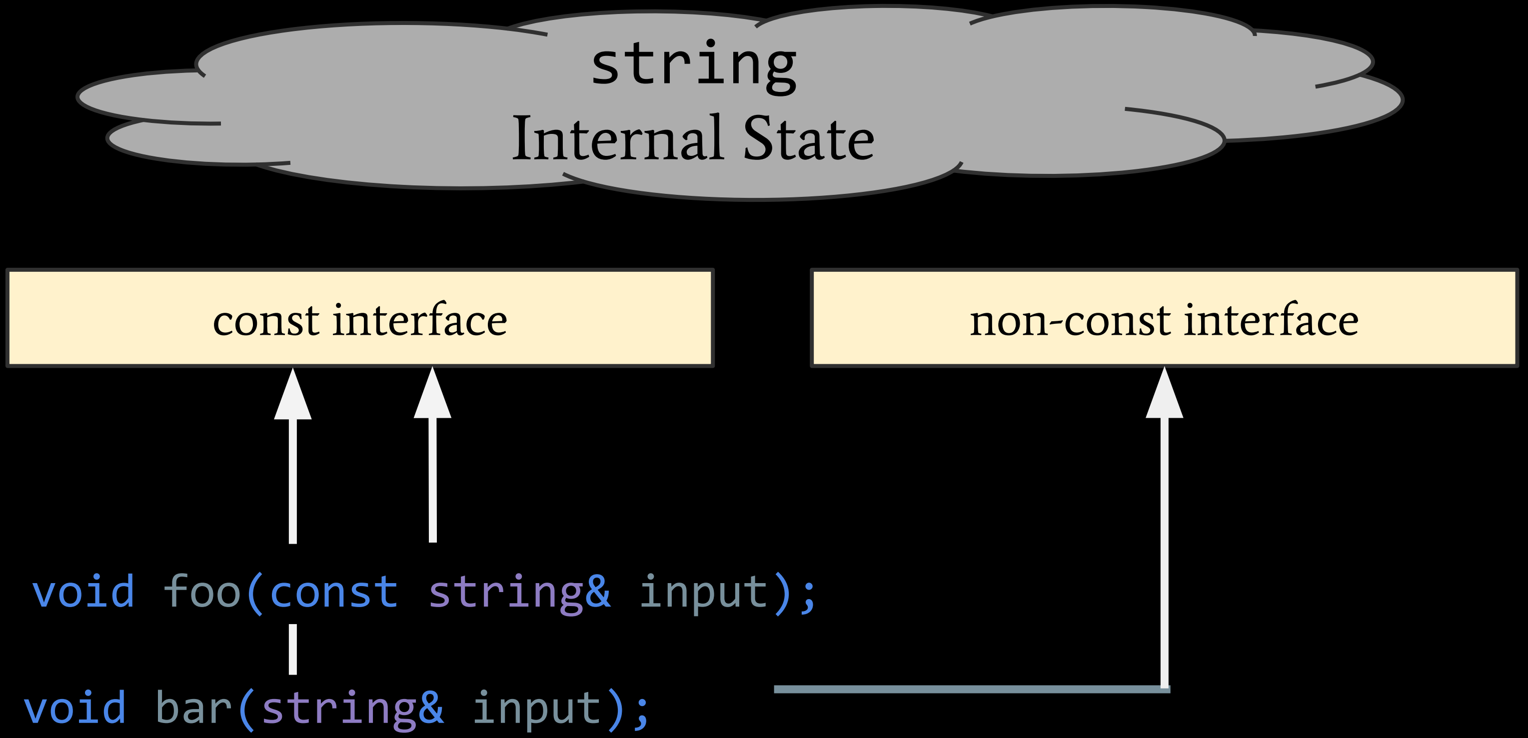
const instances of the class have to go through const interface

Non-const instance of the class can go through either

```
size_t string::size() const {
    // implementation
}

void string::clear() {
    // implementation
}
```

Const Interface



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`

Templatised 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.

```
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.

```
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.

```
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.

```
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.

```
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
}
```

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 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
}
```

Tell compiler `ValueType` is a generic type.

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

```
template <typename ValueType>
class Vector {

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

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


}
```

Compiler error

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) {


}
```

Compiler error

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) {


}
```

All good!

Class Templates - Details II

Must use `typename` keyword for nested types:

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


}
```

Class Templates - Details II

Must use `typename` keyword for nested types:

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


}
```

```
missing 'typename' prior to dependent type name 'Vector<ValueType>::iterator'
Vector<ValueType>::iterator Vector<ValueType>::begin() {
^~~~~~~~~~~~~~~~~~~~~~~~~~~
typename
/Users/alimalik/Desktop/Programs/C++/StringVector/strvector.h
```

Class Templates - Details II

Must use `typename` keyword for nested types:

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


}
```

Class Templates - Details II

Must use `typename` keyword for nested types:

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


}
```

Next Time

Const Correctness