

Welcome back! Link to Attendance Form ↓



Last Time: Classes

- What are classes?
 - Turn to the person next to you and think of one thing you remember from Tuesday's lecture!
- A class bundles data and methods for an object together

A Point on classes

```
class Point {  
public:  
    Point(int x, int y),  
    ~Point();  
    int getX();  
    int getY();  
    void setX();  
    void setY();  
  
private:  
    int x;  
    int y;  
};
```

public:

Accessible by everyone!

constructor

Initializes this class

destructor

Cleans up this class
Usually don't need this

private:

Only visible to us!
Implementation details

A Point on classes

```
class Point {  
public:  
    Point(int x, int y);  
    ~Point();  
    int getX();  
    int getY();  
  
private:  
    int x;  
    int y;  
    std::string color;  
}
```

Point.h (header file)

Contains **interface, declarations**

```
#include "Point.h"  
  
Point::Point(int x, int y)  
    : x(x), y(y) {}  
  
int Point::getX() {  
    return x;  
}  
  
int Point::getY() {  
    return y;  
}
```

Point.cpp

Contains **implementation, definitions**

What is `this`?

It's a `pointer` to the current class instance

The importance of **this**

```
int Point::getX() {  
    return x;  
}
```

```
int Point::getX() {  
    return this->x;  
}
```



These are the same

The importance of **this**

```
void Point::setX(int x)
{
    x = x;
}
```

```
void Point::setX(int x)
{
    this->x = x;
}
```

✗ Not the same

What is **this**?

```
int Point::setX(int x)
{
    this->x = x;
}
```

`Point* this`



`->`

Mwahahaha pointer dereference



What questions do you have?



bjarne_about_to_raise_hand

Last Time: **IntVector**

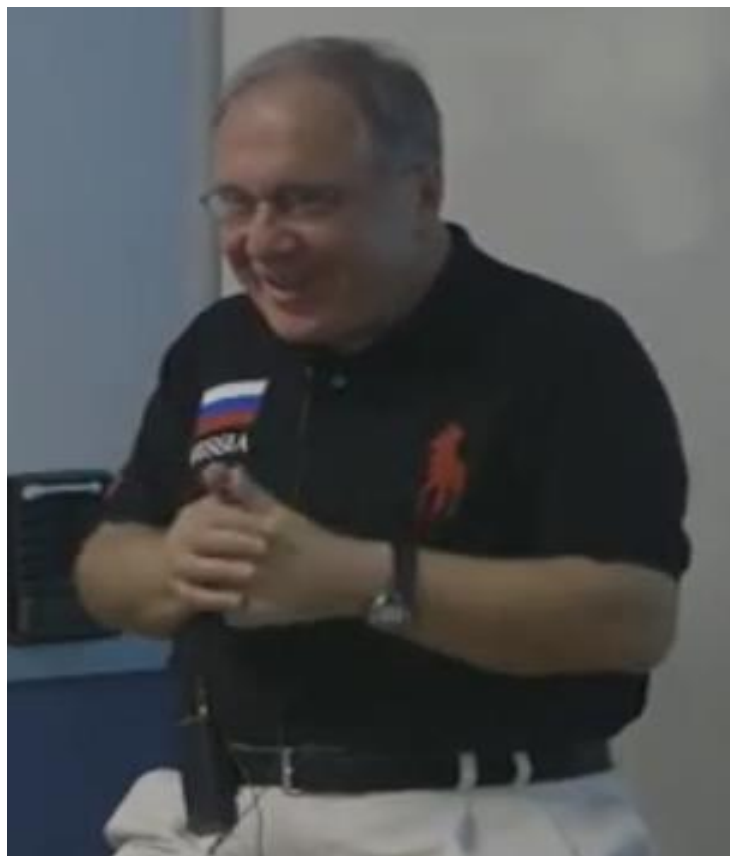
```
// Implements a sequence of strings
class IntVector {
public:
    IntVector();
    ~IntVector();

    size_t size();
    bool empty();

    void push_back(const int& elem);
    int& operator[](size_t index);
};
```

Not so fast...

You realize you need to handle...



Alexander Stepanov
Creator of STL

Vector of **doubles**?

Vector of **std::string**?

Vector of **vector of strings**?

Vector of **custom type I
haven't even thought of yet?**

IntVector is too specific

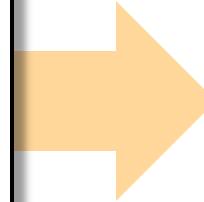
```
// Implements a sequence of strings
class IntVector {
public:
    IntVector();
    ~IntVector();

    size_t size();
    bool empty();

    void push_back(const int& elem);
    int& operator[](size_t index);
};
```

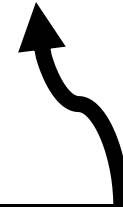
IntVector is too specific

```
class IntVector {  
    // ...  
}  
  
class DoubleVector {  
    // ...  
}  
  
class StringVector {  
    // Code to store  
    // a list of  
    // strings...  
};
```



```
template <typename T>  
class vector {  
    // So satisfying.  
};  
  
vector<int> v1;  
vector<double> v2;  
vector<string> v3;
```

std::vector<T>



How does this <T>
stuff work?

Lecture 8:

Template Classes

CS106L, Winter 2025

Today's Agenda

- Template Classes
 - How can we generalize across different types?
- Code Demo
 - Implementing a template class
- Const Correctness
 - Unlocking the power of `const`

What questions do you have?



bjarne_about_to_raise_hand

Template Classes

Templates: A bit of history

```
class IntVector {
```

```
class DoubleVector {
```

```
class StringVector {
```

```
    // Code to store
```

```
    // a list of
```

```
    // strings...
```

```
};
```

Templates: A bit of history

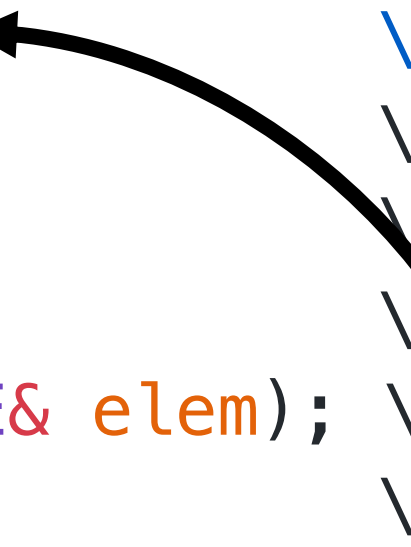
```
class IntVector {  
public:  
    int& at(size_t index);  
    void push_back(const int& elem);  
private:  
    int* elems;  
    size_t logical_size;  
    size_t array_size;  
};
```

Templates: A bit of history

```
class IntVector {  
public:  
    int& at(size_t index);  
    void push_back(const int& elem);  
private:  
    int* elems;  
    size_t logical_size;  
    size_t array_size;  
};
```

Templates: A bit of history

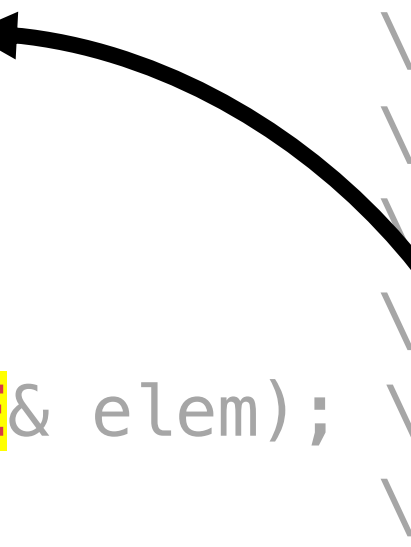
```
#define GENERATE_VECTOR(MY_TYPE)
class MY_TYPE##Vector {
public:
    MY_TYPE& at(size_t index);
    void push_back(const MY_TYPE& elem);
private:
    MY_TYPE* elems;
    size_t logical_size;
    size_t array_size;
};
```



Preprocessor Macro
Runs before compiler

Templates: A bit of history

```
#define GENERATE_VECTOR(MY_TYPE)
class MY_TYPE##Vector {
public:
    MY_TYPE& at(size_t index);
    void push_back(const MY_TYPE& elem);
private:
    MY_TYPE* elems;
    size_t logical_size;
    size_t array_size;
};
```



Preprocessor Macro
Runs before compiler

Templates: A bit of history

```
#include "grandmas_template.h"
```

```
GENERATE_VECTOR(int)
```

```
intVector v1;
```

```
v1.push_back(5);
```

Code generation!!!

Depending on what type
we pass in, we get a
different vector!

Templates: A bit of history

```
#include "grandmas_template.h"

class intVector {
public:
    int& at(size_t index);
    void push_back(const int& elem);
private:
    int* elems;
    size_t logical_size;
    size_t array_size;
};

intVector v1;
v1.push_back(5);
```

Code generation!!!

Depending on what type
we pass in, we get a
different vector!

Templates: A bit of history

```
#include "grandmas_template.h"

class intVector {
public:
    int& at(size_t index);
    void push_back(const int& elem);
private:
    int* elems;
    size_t logical_size;
    size_t array_size;
};

intVector v1;
v1.push_back(5);
```

Problems with macros


- Clunky syntax
- Hard to type check
- What if you forget to call macro?
 - Or call it more than once?



Key Idea: Templates automate code generation

Templates have come a long way

```
template <typename T>  
class Vector {  
public:  
    T& at(size_t index);  
    void push_back(const T& elem);  
private:  
    T* elems;  
};
```



Template Declaration

`Vector` is a template that takes in *the name of a type* `T`

`T` gets replaced when `Vector` is **instantiated**

Template Instantiation

```
Vector<int> intVec;  
Vector<double> doubleVec;  
Vector<std::string> strVec;  
  
Vector<Vector<int>> vecVec;  
  
struct MyCustomType {};  
Vector<MyCustomType> structVec;
```

Template Instantiation

Code for a specific type is generated on-demand, when you use it

Template Instantiation

When you write code like this...

```
template <typename T>
class Vector {
    T& at(size_t index);
    // More methods...
};
```

```
Vector<int> v;
```

Compiler produces code like this...

```
class IntVector {
    int at(size_t index);
    // More methods...
};
```

```
IntVector v;
```


A template is like a factory

int

string



Vector<int>

Vector<string>

```
template <typename T>  
class Vector
```

What questions do you have?



bjarne_about_to_raise_hand

Templates vs. Types

```
template <typename T>  
class Vector
```

This is a template.
It's **not** a type

```
Vector<std::string>
```

This is a type.
A.K.A a template
instantiation

Templates vs. Types

The template



The type

`Vector<string>`

```
template <typename T>  
class Vector
```

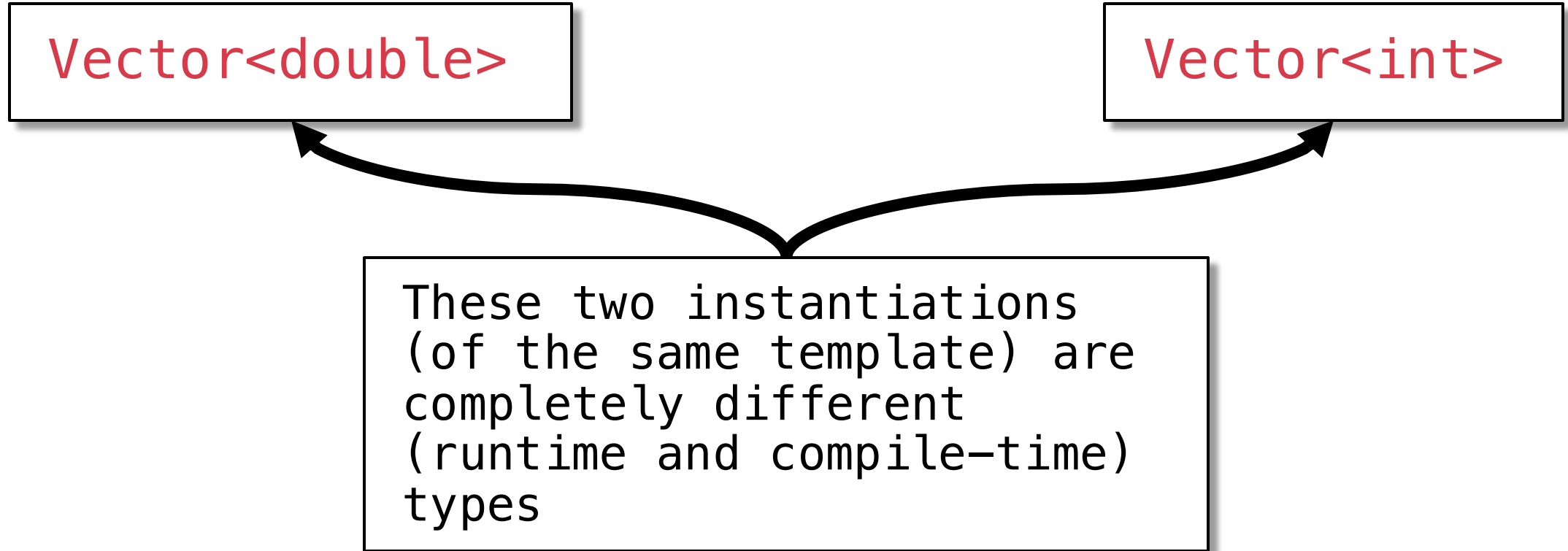
What's the problem with this code?

```
void foo(std::vector<int> v);

int main() {
    std::vector<double> v;
    foo(v);
}
```

✗ No suitable user-defined conversion from "std::vector<double>" to "std::vector<int>" exists

Note: These are two **distinct types**



Food for thought: compare this to a language like Java where an `ArrayList<int>` and `ArrayList<double>` share the same runtime type.

Fun Fact: **typename** is interchangeable

```
template <typename T>  
class Vector{};
```

```
template <size_t N>  
class SizeTemplate {};
```

```
SizeTemplate<5> s;
```

```
template <bool B>  
class BoolTemplate {};
```

```
BoolTemplate<true> b;
```

Fun Fact: **typename** is interchangeable

```
template<typename T, std::size_t N>
struct std::array { /* ... */ };

// An array of exactly 5 strings
std::array<std::string, 5> arr;
```

Why use an **array** over **vector**? It avoids heap allocations.

The compiler will know exactly how much space an **array<string, 5>** takes (the size is baked into the type!), allowing it to be stack allocated

What questions do you have?



bjarne_about_to_raise_hand

 **A few template quirks** 

(1) Must copy `template <...>` syntax in `.cpp`

Template class implementation

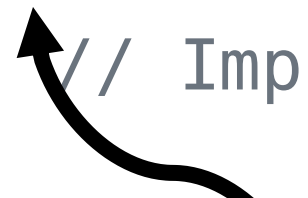
When implementing a template, you might try something like this

```
// Vector.h
```

```
template <typename T>  
class Vector {  
public:  
    T& at(size_t i);  
};
```

```
// Vector.cpp
```

```
T& Vector::at(size_t i) {  
    // Implementation...  
}
```



Compiler: “I don’t know what **T** is!”

Template class implementation

When implementing a template, must copy over template declaration

```
// Vector.cpp  
  
template <typename T>  
T& Vector::at(size_t i) {  
    // Implementation...  
}
```

Does anyone still see
a problem with this?

Template class implementation

`Vector` is not a type, but `Vector<T>` is

```
// Vector.cpp

template <typename T>
T& Vector<T>::at(size_t i) {
    // Implementation...
}
```

Compiler: “Ahh... I’m
happy now 😊😊”

(2) .h must include .cpp at bottom of file

Normal class implementation

For non-template classes, the `.cpp` file includes the `.h` file

```
// StrVector.h

class StrVector {
public:
    string& at(size_t i);
};
```

```
// StrVector.cpp

#include "StrVector.h"

string& StrVector::at(size_t i)
{
    // Implementation...
}
```


Template class implementation

For template classes, the `.h` file includes the `.cpp` file

```
// Vector.h

template <typename T>
class Vector {
public:
    T& at(size_t i);
};
```

```
#include "Vector.cpp"
```

```
// Vector.cpp

template <typename T>
T& Vector<T>::at(size_t i) {
    // Implementation...
}
```

That's pretty weird 🤪 Why?

- Template `.h` must include `.cpp` due to the way template code generation is implemented in the compiler (and linker)
- Don't worry too much about the *why* (unless you're curious!)
- There are ways to get around this (ask us after!)

(3) `typename` is the same as `class`

(3) **typename** is the same as **class**

```
template <typename T>  
class Vector{};
```

```
template <class T>  
class Vector{};
```

(3) **typename** is the same as **class**

All of the following are identical:

```
template <typename K, typename V>  
struct pair;
```

```
template <class K, class V>  
struct pair;
```

```
template <class K, typename V>  
struct pair;
```

```
template <class K, typename V>  
struct pair;
```

What questions do you have?



bjarne_about_to_raise_hand

Code Demo

Let's implement `Vector<T>`

Let's code this together 

106l.vercel.app/vector

What questions do you have?



bjarne_about_to_raise_hand

Const Correctness

Let's use our **Vector** class!

```
void printVec(const Vector<int>& v) {  
    for (size_t i = 0; i < v.size(); i++) {  
        std::cout << v.at(i) << "  
    }  
    std::cout << std::endl;  
}
```

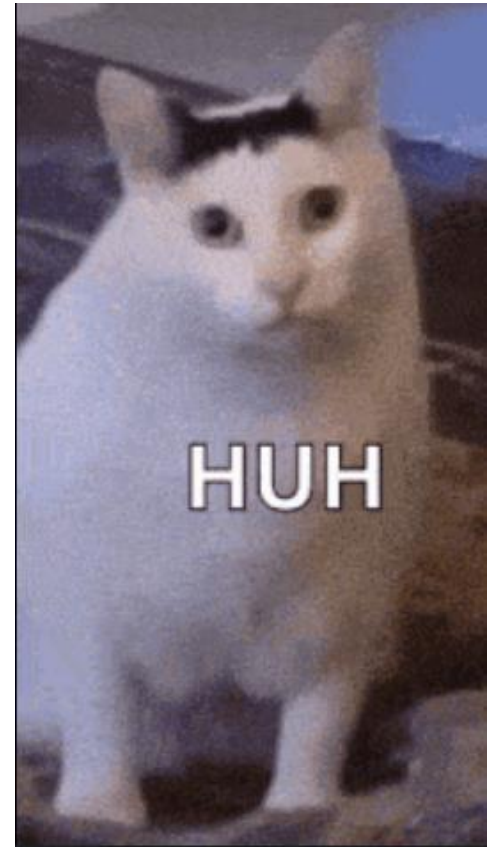


Compiler: "No such
method `size`!"

Huh? But there is a method called **size**

```
template<class T>
class Vector {
public:
    size_t size();
    bool empty();

    T& operator[] (size_t index);
    T& at(size_t index);
    void push_back(const T& elem);
};
```



What is the problem?

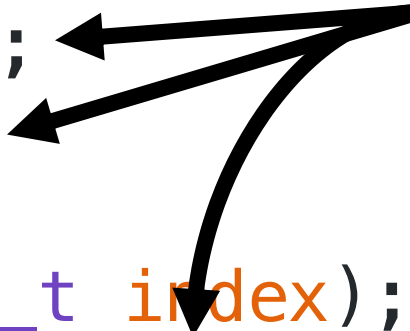
```
void printVec(const Vector<int>& v) {  
    for (size_t i = 0; i < v.size(); i++) {  
        std::cout << v.at(i) << " ";  
    }  
    std::cout << std::endl;  
}
```

- By passing **v** as **const**, we promise not to modify **v**
- Compiler cannot be sure if methods like **size** and **at** will modify **v**
- Remember, member functions *can* access member variables

How do we fix it?

```
template<class T>
class Vector {
public:
    size_t size() const;
    bool empty() const;

    T& operator[] (size_t index);
    T& at(size_t index) const;
    void push_back(const T& elem)
};
```



const method:

"Dear compiler,

I promise not to
modify this object
inside of this
method. Please hold
me accountable.

Love, Jacob 🥰"

How do we fix it (.cpp file)?

```
template <class T>
size_t Vector<T>::size() const {
    return logical_size;
}

// Other methods...
```

Make sure to also add **const** to the implementation, or the compiler will scream

How do we fix it (.cpp file)?

```
template <class T>
size_t Vector<T>::size() const {
    this->logical_size = 106; // 🤡🤡🤡
    return logical_size;
}
```

```
// error: cannot assign to rvalue
// within const member function
```

Inside a **const** method, **this** has type **const Vector<T>***

What is **this**?

```
void Point::setX(int x)
{
    this->x = x;
}
```

Point* this



```
void Point::getX(int x)
const
{
    return this->x;
}
```

const Point* this



The **const** interface

- Objects marked as **const** can only make use of the **const interface**
- The **const** interface are the functions that are **const** in an object

The **const** interface

```
template<class T>
class Vector {
public:
    size_t size() const;
    bool empty() const;
    void push_back(const T& elem);
private:
    size_t logical_size;
    T* elems;
};
```

Vector<T>

```
template<class T>
class Vector {
public:
    size_t size() const;
    bool empty() const;
    void push_back(const T& elem);
private:
    const size_t logical_size;
    const T* elems;
};
```

const Vector<T>

What questions do you have?



bjarne_about_to_raise_hand

Back to our **Vector** class!

```
void printVec(const Vector<int>& v) {  
    for (size_t i = 0; i < v.size(); i++) {  
        std::cout << v.at(i) << " ";  
    }  
    std::cout << std::endl;  
}
```

Compiler: “🐻 **const Vector<int>**
has no **size, at!!!**”

Back to our **Vector** class!

```
template<class T>
class Vector {
public:
    size_t size();
    bool empty();

    T& operator[] (size_t index);
    T& at(size_t index);
    void push_back(const T& elem);
};
```

Let's add **const** to the methods which don't modify **Vector**

Back to our **Vector** class!

```
template<class T>
class Vector {
public:
    size_t size() const;
    bool empty() const;

    T& operator[] (size_t index);
    T& at(size_t index) const;
    void push_back(const T& elem);
};
```

Let's add **const** to the methods which don't modify **Vector**

Back to our **Vector** class!

```
void printVec(const Vector<int>& v) {  
    for (size_t i = 0; i < v.size(); i++) {  
        std::cout << v.at(i) << " ";  
    }  
    std::cout << std::endl;  
}
```

Compiler: “ Everything looks good to me!”

Back to our **Vector** class!

```
template<class T>
class Vector {
public:
    size_t size() const;
    bool empty() const;

    T& operator[] (size_t index);
    T& at(size_t index) const;
    void push_back(const T& elem);
};
```

There's at least **one**
(or maybe two)
problems with how this
method is declared.

Turn to a partner and
take 60s to talk about
why!



Problem #1: **const** consumers can modify!

Since we return a **non-const reference**, we can assign to it!

```
T& at(size_t index) const;

void oops(const Vector<int>& v) {
    v.at(0) = 42;
}
```

Remember, since **v** is const, we shouldn't be able to modify it

Solution: return a **const** reference

```
template<class T>
class Vector {
public:
    size_t size() const;
    bool empty() const;

    T& operator[] (size_t index);
    const T& at(size_t index) const;
    void push_back(const T& elem);
};
```

Hmm... There's still a problem here

Problem #2: non-**const** consumers can't modify!

If we return a const reference, now we cannot update elements!

```
const T& at(size_t index) const;

void ooh(Vector<int>& v) {
    v.at(0) = 42;
}
```

✗ Can't assign to **const int&**

Solution: **const** overloading!

- Let's define two versions of our **at** method
- One version gets called for **const** instances
- ...And another that gets called for non-**const** instances

```
template<class T>
class Vector {
public:
    const T& at(size_t index) const;
    T& at(size_t index);
};
```

Solution: **const** overloading (.cpp file)!

```
template <class T>
const T& Vector<T>::at(size_t index) const {
    return elems[index];
}
```

```
template <class T>
T& Vector<T>::at(size_t index) {
    return elems[index];
}
```

What questions do you have?



bjarne_about_to_raise_hand

Solution: **const** overloading (.cpp file)!

```
template <class T>
const T& Vector<T>::at(size_t index) const {
    return elems[index];
}
```

```
template <class T>
T& Vector<T>::at(size_t index)
    return elems[index];
}
```

Two methods with the same implementation. It's a bit redundant, but it's only one line

What if we added a **findElement**?

```
template<class T>
class Vector {
public:
    T& at(size_t index);
    const T& at(size_t index) const;
    T& findElement(const T& value);
    const T& findElement(const T& value) const;
};
```

Implementing findElement

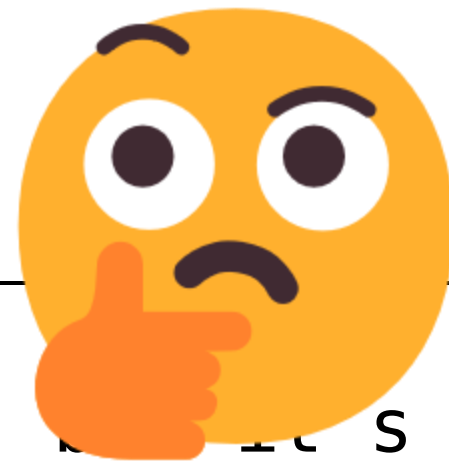
```
template <typename T>
T& Vector<T>::findElement(const T& value) {
    for (size_t i = 0; i < logical_size; i++) {
        if (elems[i] == elem) return elems[i];
    }
    throw std::out_of_range("Element not found");
}
```

// What about the const version of findElement?

Implementing findElement

```
template <typename T>
T& Vector<T>::findElement(const T& value) {
    for (size_t i = 0; i < logical_size; i++) {
        if (elems[i] == elem) return elems[i];
    }
    throw std::out_of_range("Element not found");
}

template <typename T>
const T& Vector<T>::findElement(const T& value) {
    for (size_t i = 0; i < logical_size; i++) {
        if (elems[i] == elem) return elems[i];
    }
    throw std::out_of_range("Element not found");
}
```



This works, but it's super redundant. There must be a better way!

A slight (but useful) aside

- Casting: the process of converting one type to another
 - There are *many* ways to cast in C++
- `const_cast` allows us to “cast away” the `const`-ness of a variable
 - Usage: `const_cast<target_type>(expression)`
 - So why is this useful?

Implementing findElement

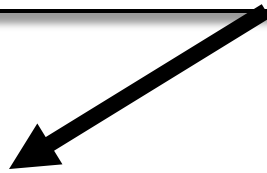
```
template <typename T>
T& Vector<T>::findElement(const T& value) {
    for (size_t i = 0; i < logi
        if (elems[i] == elem) ret
    }
    throw std::out_of_range("El
}
```

Ahh no more
redundancy... But what
in the Bjarne is going
on here?

```
template <typename T>
const T& Vector<T>::findElement(const T& value) const {
    return const_cast<Vector<T>&>(*this).findElement(value);
}
```

```
const_cast<Vector<T>&>(*this).findElement(value);
```

`const_cast` casts away
the `const`



```
const_cast<Vector<T>&>(*this).findElement(value);
```


`const_cast` casts away
the `const`

`*this` dereferences a
`const Vector<T>*`,
giving us a `const-ref`

`const_cast<Vector<T>&>(*this).findElement(value);`

`const Vector<T>&`

`const_cast` casts away
the `const`

`*this` dereferences a
`const Vector<T>*`,
giving us a `const-ref`

`const_cast<Vector<T>&>(*this).findElement(value);`

`Vector<T>&` is a
non-const reference,
the type we would like

`const_cast` casts away
the `const`

`*this` dereferences a
`const Vector<T>*`,
giving us a `const-ref`

`const_cast<Vector<T>&>(*this).findElement(value);`

`Vector<T>&` is a
non-const reference,
the type we would like

Phew... This is the
non-const version of
`findElement`

`const_cast` casts away
the `const`

`*this` dereferences a
`const Vector<T>*`,
giving us a const-ref

`const_cast<Vector<T>&>(*this).findElement(value);`

`Vector<T>&` is a
non-const reference,
the type we would like

Phew... This is the
non-const version of
`findElement`

const_cast forces compiler to pick right overload

```
template<class T>
class Vector {
public:
    T& at(size_t index);
    const T& at(size_t index) const;
    T& findElement(const T& value);
    const T& findElement(const T& value) const;
};
```

Implementing findElement

```
template <typename T>
T& Vector<T>::findElement(const T& value) {
    for (size_t i = 0; i < logical_size; i++) {
        if (elems[i] == elem) return elems[i];
    }
    throw std::out_of_range("Element not found");
}

template <typename T>
const T& Vector<T>::findElement(const T& value) const {
    return const_cast<Vector<T>&>(*this).findElement(value);
}
```

When to use `const_cast`?

- Short answer: just about never
- `const_cast` tells the compiler: “don’t worry I’ve got this”
- If you need a mutable value, just don’t add `const` in the first place
- Valid uses of `const_cast` are few and far between

What questions do you have?



bjarne_about_to_raise_hand

const_cast makes an *entire* object mutable

Is there anything more fine-grained?

A C++ party trick: **mutable** keyword

Like **const_cast**, **mutable** circumvents const protections. Use it carefully!

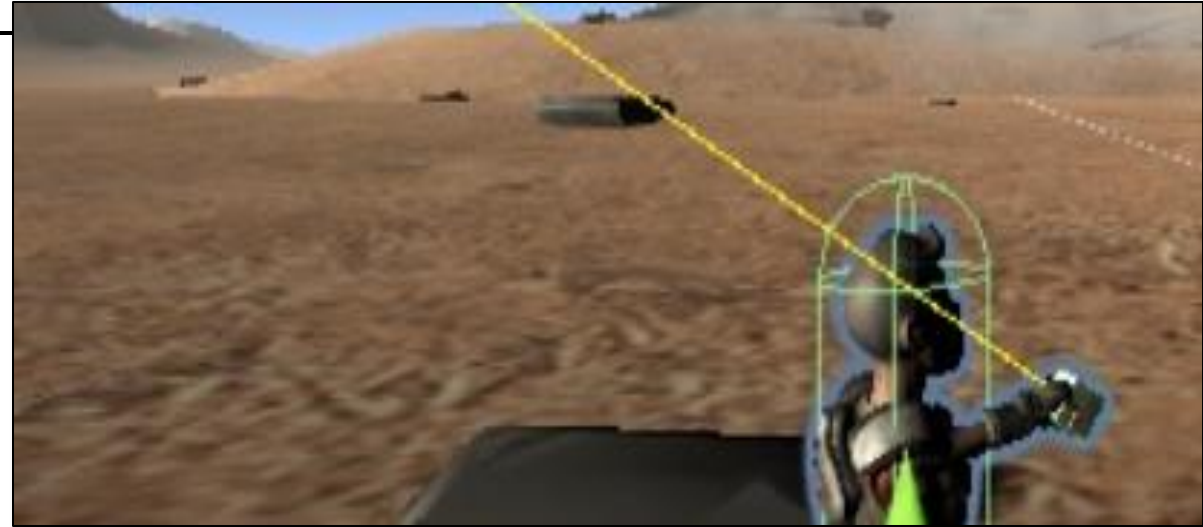
```
struct MutableStruct {  
    int dontTouchThis;  
    mutable double iCanChange;  
};
```

```
const MutableStruct cm;  
// cm.dontTouchThis = 42;    // ❌ Not allowed, cm is const  
cm.iCanChange = 3.14;       // ✅ Ok, iCanChange is mutable
```

mutable example: storing debug info

```
struct CameraRay {  
    Point origin;  
    Direction direction;  
    mutable Color debugColor;  
}
```

```
void renderRay(const CameraRay& ray) {  
    ray.debugColor = Color.Yellow; // Show debug ray  
    /* Rendering logic goes here ... */  
}
```



Recap

Meme of the Day



What We Covered

- Template Classes
 - Template classes generalize logic across types!
- Code Demo
 - We implemented an `IntVector`, and then a templated `Vector`!
- Const Correctness
 - `const` makes an entire object read-only
 - Mark methods `const` when they don't modify the object
 - `const_cast` and `mutable` can circumvent compiler in *rare* cases!

Next Time: Template Functions

Unlocking the power of templates