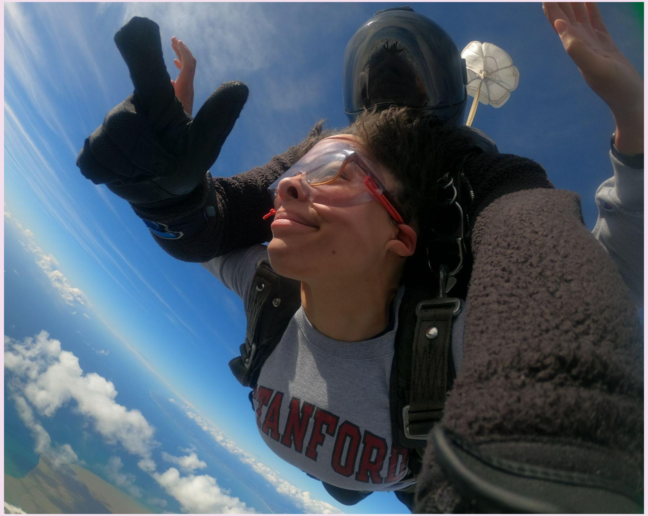
Template Classes + Const Correctness

How do we make our classes general? How do we make them safe?

Attendance

bit.ly/446GZRK

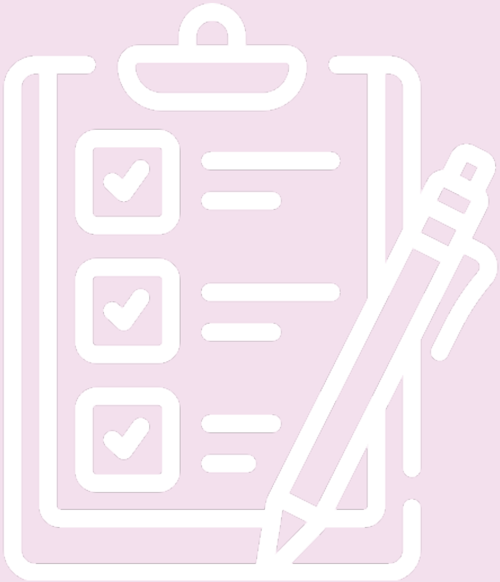


CS 106B covers the barebones of C++ classes... we'll be covering the rest

template classes • **const-correctness** • operator overloading • special member functions • move semantics • RAII

This lecture is incredibly important for assn 2
But also very dense so ask questions and we may use the full time block

Today



- **Classes Recap**
- Template Classes
- Const Correctness

Turning `Student` into a class: Header File

```
//student.h
class Student {
    public:
    std::string getName();
    void setName(string
    name);
    int getAge();
    void setAge(int age);

    private:
    std::string name;
    std::string state;
    int age;
};
```

Public section:

- Users of the Student object can directly access anything here!
- Defines an interface for interacting with the private member variables!

Private section:

- Usually contains all member variables
- Users can't access or modify anything in the private section

Turning `Student` into a class: Header File + .cpp File

//student.h

```
class Student {
    public:
    std::string getName();
    void setName(string
    name);
    int getAge();
    void setAge(int age);

    private:
    std::string name;
    std::string state;
    int age;
 };
```

//student.cpp

```
#include student.h
std::string Student::getName(){
//implementation here!
}

void Student::setName(){
}

int Student::getAge(){
}

 void Student::setAge (int age){
}
```

//student.cpp

```
#include student.h
std::string Student::getName(){
    return name; //we can access name here!
}
void Student::setName(std::string name){
    name = name; //huh?
}
int Student::getAge(){

}
 void Student::setAge(int age){

}
```

//student.h

```
class Student {
    public:
    std::string getName();
    void setName(string
    name);
    int getAge();
    void setAge(int age);

    private:
    std::string name;
    std::string state;
    int age;
 };
```

The `this` keyword!

- Here, we mean "set the Student private member variable `name` equal to the parameter `name`"
- `this->element_name` means "the item in this Student object with name *element_name*". Use `this` for resolving naming conflicts!

```
void Student::setName(string name){
    this->name = name; //better!
}
```

What if the object has member variables?

If we need to access a pointer's object's member variables, instead of dereferencing (*ptr) and then accessing (.var), there's a shorthand!

(*ptr).var == ptr->var

this is a pointer (Student *) to a Student object. Deference the pointer (*this) to get the actual object and then access the object's name private member variable (*this).name = name; or this->name = name;

```
}
void Student::setName(string name){
    this->name = name; //resolved!
}
int Student::getAge(){
    return age;
}
 void Student::setAge(int age){
    //We can define what "age" means!
    if(age >= 0){
        this -> age = age;
    }
    else error("Age cannot be negative!");
}
```

//student.h

```
class Student {
    public:
    std::string getName();
    void setName(string
    name);
    int getAge();
    void setAge(int age);

    private:
    std::string name;
    std::string state;
    int age;
 };
```

//student.cpp

```
#include student.h
std::string Student::getName(){
    return name; //we can access name here!
}
void Student::setName(string name){
    this->name = name; //resolved!
}
int Student::getAge(){
    return age;
}
 void Student::setAge(int age){
    //We can define what "age" means!
    if(age >= 0){
        this -> age = age;
    }
    else error("Age cannot be negative!");
}
```

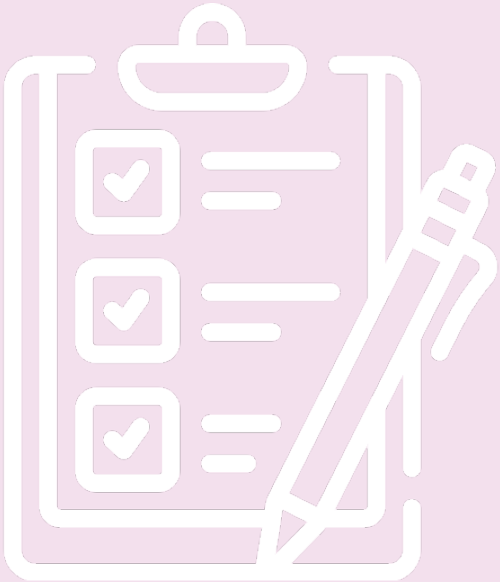
//student.h

```
class Student {
    public:
    std::string getName();
    void setName(string
    name);
    int getAge();
    void setAge(int age);

    private:
    std::string name;
    std::string state;
    int age;
 };
```

Questions?

Today



- ~~Classes Recap~~
- **Template Classes**
- Const Correctness

The problem with StrVector

- Vectors should be able to contain any data type!

The problem with StrVector

- Vectors should be able to contain any data type!

Solution? Create IntVector, DoubleVector, BoolVector etc..

The problem with StrVector

- Vectors should be able to contain any data type!

Solution? Create IntVector, DoubleVector, BoolVector etc..

- What if we want to make a vector of `Student`s?
  - How are we supposed to know about every custom class?
- What if we don't want to write a class for every type we can think of?

The problem with StrVector

- Vectors should be able to contain any data type!

~~Solution? Create IntVector, DoubleVector, BoolVector etc..~~

- What if we want to make a vector of `Student`s?
  - How are we supposed to know about every custom class?
- What if we don't want to write a class for every type we can think of?

SOLUTION: Template classes!

Definition

Template Class: A class that is parametrized over some number of types. A class that is comprised of member variables of a general type/types.

A simpler class: My Int Pair (so specific)

```
//myintpair.h
class MyPair {
    public:
        int getFirst();
        int getSecond();

        void setFirst(int f);
        void setSecond(int f);
    private:
        int first;
        int second;
};
```

Can only hold ints :(

Writing a Template Class: Syntax

//mypair.h

```
template<typename First, typename Second> class MyPair {
    public:
        First getFirst();
        Second getSecond();

        void setFirst(First f);
        void setSecond(Second f);
    private:
        First first;
        Second second;
};
```

Generic! First and Second can be any type (including custom types) not just ints 😊

Use generic typenames as placeholders!

Implementing a Template Class: Syntax

```
//mypair.cpp
#include “mypair.h”

First MyPair::getFirst(){
    return first;
}
```

Implementing a Template Class: Syntax

//mypair.cpp

```
#include “mypair.h”

First MyPair::getFirst(){
    return first;
}
//Compile error! Must announce every member function is templated :/
```

Implementing a Template Class: Syntax

//mypair.cpp

```
#include "mypair.h"

template<class First, typename Second>
First MyPair::getFirst(){
    return first;
}
// So are we good now????
```

Implementing a Template Class: Syntax

```
//mypair.cpp
#include "mypair.h"

template<class First, typename Second>
First MyPair::getFirst(){
    return first;
}
//Compile error! The namespace of the class isn't just MyPair
```

Implementing a Template Class: Syntax

//mypair.cpp

```
#include "mypair.h"

template<class First, typename Second>
First MyPair<First, Second>::getFirst(){
    return first;
}
// Fixed!
```

Implementing a Template Class: Syntax

Template Function

```
template<class First, typename Second>
First MyPair<First, Second>::getFirst(){
    return first;
}
// Fixed!
```

Implementing a Template Class: Syntax

Template Function

class and typename interchangeable

```
template<class First, typename Second>
First MyPair<First, Second>::getFirst(){
    return first;
}
// Fixed!
```

Implementing a Template Class: Syntax

Template Function

class and typename interchangeable

List of template variables

```
template<class First, typename Second>
First MyPair<First, Second>::getFirst(){
    return first;
}
// Fixed!
```

C++ reference is like 10% less scary now!

std::vector

Defined in header <vector>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
```

(1)

Less scary. Just saying that vector is a template class with 2 parameters

```
namespace pmr {
    template< class T >
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;
}
```

17)

Still scary

1) `std::vector` is a sequence container that encapsulates dynamic size arrays.

2) `std::pmr::vector` is an alias template that uses a polymorphic allocator.

Screenshot from: https://en.cppreference.com/w/cpp/container/vector

Questions?

Member Types

- Sometimes, we need a name for a type that is dependent on our template types
- Recall: iterators

```
std::vector<int> a = {1, 2};
std::vector<int>::iterator it = a.begin();
```

Member Types

- Sometimes, we need a name for a type that is dependent on our template types
- Recall: iterators

```
std::vector<int> a = {1, 2};
std::vector<int>::iterator it = a.begin();
```

- iterator is a **member type** of vector

Member Types: Syntax

```
//vector.h
template<typename T> class vector {
    public:
    using iterator = T*  // something internal like T*

    iterator begin();
}
```

Member Types: Syntax

```
//vector.h
template<typename T> class vector {
    public:
    using iterator = T*  // something internal like T*

    iterator begin();
}
```

```
//vector.cpp
template <typename T>
iterator vector<T>::begin() {...}
//compile error! Why?
```

Member Types: Syntax

```
//vector.h
template<typename T> class vector {
    public:
    using iterator = T*  // something internal like T*

    iterator begin();
}
```

```
//vector.cpp
template <typename T>
iterator vector<T>::begin() {...}
//iterator is a nested type in namespace vector<T>::
```

Member Types: Syntax

```
//vector.h
template<typename T> class vector {
    public:
    using iterator = T*  // something internal like T*

    iterator begin();
}
```

```
//vector.cpp
template <typename T>
typename vector<T>::iterator vector<T>::begin() {...}
```

Aside: Type Aliases

- You can use `using type_name = type` in application code as well!
- When using it in a class interface, it defines a nested type, like `vector::iterator`
- When using it in application code, like main.cpp, it just creates another name for `type` within that scope (until the next unmatched })

Member Types: Summary

- Used to make sure your clients have a standardized way to access important types.
- Lives in your namespace: **vector<T>::iterator**.
- After class specifier, you can use the alias directly (e.g. inside function arguments, inside function body).
- Before class specifier, use **typename.**

realVector.cpp

No more “this is the simplified version of the real thing”… We are writing the real thing (just a little simplified :p)

Recap: Template classes

- Add `template<class T1, T2..>` before class definition in .h

Recap: Template classes

- Add `template<class T1, T2..>` before class definition in .h
- Add `template<class T1, T2..>`before all function signatures in .cpp

Recap: Template classes

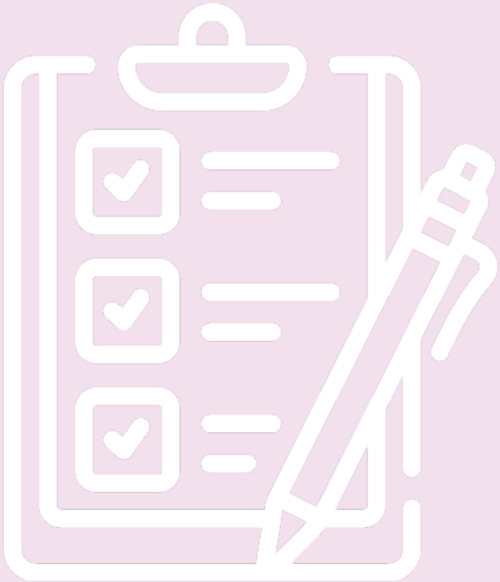
- Add `template<class T1, T2..>` before class definition in .h
- Add `template<class T1, T2..>`before all function signatures in .cpp
- When returning nested types (like iterator types), put `typename ClassName<T1, T2..>::member_type` as return type, not just `member_type`

Recap: Template classes

- Add `template<class T1, T2..>` before class definition in .h
- Add `template<class T1, T2..>`before all function signatures in .cpp
- When returning nested types (like iterator types), put `typename ClassName<T1, T2..>::member_type` as return type, not just `member_type`
- Templates don't emit code until instantiated, so `#include` the .cpp file in the .h file, not the other way around!

Questions?

Today



- ~~Finish StrVector~~
- ~~Template Classes~~
- **Const Correctness**

Const and Const References

Definition

`const`: **keyword indicating a variable, function or parameter can't be modified**

`const` indicates a variable can't be modified!

`const` variables can be references or not!

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};  // a const variable
std::vector<int>& ref = vec;          // a regular reference
const std::vector<int>& c_ref = vec;  // a const reference

vec.push_back(3);
c_vec.push_back(3);
ref.push_back(3);
c_ref.push_back(3);
```

`const` indicates a variable can't be modified!

`const` variables can be references or not!

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};  // a const variable
std::vector<int>& ref = vec;         // a regular reference
const std::vector<int>& c_ref = vec;  // a const reference

vec.push_back(3);    // OKAY
c_vec.push_back(3);
ref.push_back(3);
c_ref.push_back(3);
```

`const` indicates a variable can't be modified!

`const` variables can be references or not!

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};  // a const variable
std::vector<int>& ref = vec;         // a regular reference
const std::vector<int>& c_ref = vec;  // a const reference

vec.push_back(3);    // OKAY
c_vec.push_back(3);  // BAD - const
ref.push_back(3);
c_ref.push_back(3);
```

`const` indicates a variable can't be modified!

`const` variables can be references or not!

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};  // a const variable
std::vector<int>& ref = vec;         // a regular reference
const std::vector<int>& c_ref = vec;  // a const reference

vec.push_back(3);     // OKAY
c_vec.push_back(3);   // BAD - const
ref.push_back(3);    // OKAY
c_ref.push_back(3);
```

`const` indicates a variable can't be modified!

`const` variables can be references or not!

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};   // a const variable
std::vector<int>& ref = vec;          // a regular reference
const std::vector<int>& c_ref = vec;  // a const reference

vec.push_back(3);     // OKAY
c_vec.push_back(3);   // BAD - const
ref.push_back(3);    // OKAY
c_ref.push_back(3); // BAD - const
```

Why const?

Why const? Find the typo in this code

```
void f(int x, int y) {
    if ((x==2 && y==3)||(x==1))
        cout << 'a' << endl;
    if ((y==x-1)&&(x==-1||y=-1))
        cout << 'b' << endl;
    if ((x==3)&&(y==2*x))
        cout << 'c' << endl;
}
```

Why const? Find the typo in this code

```
void f(const int x, const int y) {
    if ((x==2 && y==3)||(x==1))
        cout << 'a' << endl;
    if ((y==x-1)&&(x==-1||y=-1))
        cout << 'b' << endl;
    if ((x==3)&&(y==2*x))
        cout << 'c' << endl;
}
```

y will be set to -1 whenever x != -1, and the if condition will be true whenever y equals x - 1. If x is -1, the y = -1 part won't be evaluated due to short-circuit evaluation of || operator, and y won't be changed.

Const and Classes

Recall: `Student` class

//student.h

```
class Student {
    public:
    std::string getName();
    void setName(string name);
    int getAge();
    void setAge(int age);

    private:
    std::string name;
    std::string state;
    int age;
};
```

//student.cpp

```
#include student.h
std::string Student::getName(){
    return name; //we can access name here!
}
void Student::setName(string name){
    this->name = name; //resolved!
}
int Student::getAge(){
    return age;
}
 void Student::setAge(int age){
    //We can define what “age” means!
    if(age >= 0){
        this -> age = age;
    }
    else error("Age cannot be negative!");
}
```

//student.h

```
class Student {
    public:
    std::string getName();
    void setName(string
    name);
    int getAge();
    void setAge(int age);

    private:
    std::string name;
    std::string state;
    int age;
 };
```

Using a `const Student`

```
//main.cpp
std::string stringify(const Student& s){
    return s.getName() + " is " + std::to_string(s.getAge()) +
                                              " years old." ;
}
```

Using a `const Student`

```
//main.cpp
std::string stringify(const Student& s){
    return s.getName() + " is " + std::to_string(s.getAge()) +
                                                " years old." ;
}
//compile error!
```

Using a `const Student`

```cpp
//main.cpp
std::string stringify(const Student& s){
    return s.getName() + " is " + std::to_string(s.getAge()) +
                                                    " years old." ;
}
//compile error!
```

- The compiler doesn't know `getName` and `getAge` don't modify s!

Using a `const Student`

//main.cpp

```
std::string stringify(const Student& s){
    return s.getName() + " is " + std::to_string(s.getAge()) +
                                                  " years old." ;
}
//compile error!
```

- The compiler doesn't know `getName` and `getAge` don't modify s!
- We need to promise that it doesn't by defining them as **const functions**

Using a `const Student`

```
//main.cpp
std::string stringify(const Student& s){
    return s.getName() + " is " + std::to_string(s.getAge()) +
                                                " years old." ;
}
//compile error!
```

- The compiler doesn't know `getName` and `getAge` don't modify s!
- We need to promise that it doesn't by defining them as **const functions**
- Add `const` to the **end** of function signatures!

Making Student const-correct

//student.cpp

```
#include student.h
std::string Student::getName()const{
    return name;
}
void Student::setName(string name){
    this->name = name;
}
int Student::getAge()const{
    return age;
}
 void Student::setAge(int age){
    if(age >= 0){
        this -> age = age;
    }
    else error("No Negative Age!");
}
```

//student.h

```
class Student {
    public:
    std::string getName() const;
    void setName(string name);
    int getAge() const;
    void setAge(int age);

    private:
    std::string name;
    std::string state;
    int age;
 };
```

Definition

const-interface: All member functions marked `const` in a class definition. Objects of type `const ClassName` may only use the const-interface.

Using a `const Student`

```cpp
//main.cpp
std::string stringify(const Student& s){
    return s.getName() + " is " + std::to_string(s.getAge()) +
                                              " years old." ;
}
// With our new const-interface, this will work :)
```

Making `StrVector`'s const-interface

```
class StrVector {
public:
    using iterator = std::string*;
    const size_t kInitialSize = 2;
    /*...*/
    size_t size();
    bool empty();
    void push_back(const std::string& elem);
    std::string& at(size_t indx); // like vec[] but with error checking


    iterator begin();
    iterator end();
    /*...*/
```

Making `StrVector`'s const-interface

```
class StrVector {
public:
    using iterator = std::string*;
    const size_t kInitialSize = 2;
    /*...*/
    size_t size();
    bool empty();
    void push_back(const std::string& ele );
    std::string& at(size_t indx); // like vec[] but with error checking


    iterator begin();
    iterator end();
    /*...*/
```

Questions to ask:

1.) Should this function be able available to a const object?

   a.) Can I mark the function const as is (i.e. the function doesn't modify the object)?

   b.) Otherwise, can I make a const version of the function?

Making `StrVector`'s const-interface

```
class StrVector {
public:
    using iterator = std::string*;
    const size_t kInitialSize = 2;
    /*...*/
    size_t size();
    bool empty();
    void push_back(const std::string& elem);
    std::string& at(size_t indx); // like vec[] but with error checking


    iterator begin();
    iterator end();
    /*...*/
```

Questions to ask:

1.) Should this function be able available to a const object?
   a.) Can I mark the function const as is (i.e. the function doesn't modify the object)?
   b.) Otherwise, can I make a const version of the function?

Making StrVector's const-interface

```
class StrV...
public:
    using ...
    const ...
    /*...*/
    size_t size();
    bool empty();
    void push_back(const std::string& elem);
    std::string& at(size_t indx); // like vec[] but with error checking


    iterator begin();
    iterator end();
    /*...*/
```

```
size_t StrVector::size() {
    return logicalSize;
}
```

Questions to ask:

1.) Should this function be able available to a const object?
   a.) Can I mark the function const as is (i.e. the function doesn't modify the object)?
   b.) Otherwise, can I make a const version of the function?

Making `StrVector`'s const-interface

```
class StrVector {
public:
    using iterator = std::string*;
    const size_t kInitialSize = 2;
    /*...*/
    size_t size() const;
    bool empty();
    void push_back(const std::string& elem);
    std::string& at(size_t indx); // like vec[] but with error checking


    iterator begin();
    iterator end();
    /*...*/
```

Making `StrVector`'s const-interface

```
class StrVector {
publi
    u                                                          *;
    c                                                          2;
    /
    size_t size() const;
    bool empty();
    void push_back(const std::string& elem);
    std::string& at(size_t indx); // like vec[] but with error checking


    iterator begin();
    iterator end();
    /*...*/
```

```
bool StrVector::empty() {
    return size() == 0;
}
```

Questions to ask:

1.) Should this function be able available to a const object?
   a.) Can I mark the function const as is (i.e. the function doesn't modify the object)?
   b.) Otherwise, can I make a const version of the function?

Making StrVector's const-interface

```
class StrVector {
public:
    using iterator = std::string*;
    const size_t kInitialSize = 2;
    /*...*/
    size_t size() const;
    bool empty() const;
    void push_back(const std::string& elem);
    std::string& at(size_t indx); // like vec[] but with error checking


    iterator begin();
    iterator end();
    /*...*/
```

Making `StrVector`'s const-interface

```
class StrVector {
public:
    using iterator = std::string*;
    const size_t kInitialSize = 2;
    /*...*/
    size_t size() const;
    bool empty() const;
    void push_back(const std::string& elem);
    std::string& at(size_t indx); // like vec[] but with error checking


    iterator begin();
    iterator end();
    /*...*/
```

Questions to ask:

1.) Should this function be able available to a const object?
   a.) Can I mark the function const as is (i.e. the function doesn't modify the object)?
   b.) Otherwise, can I make a const version of the function?

face

```
void StrVector::push_back(const std::string& elem)
{
    if (allocatedSize == size()) {
        grow();
    }
    *end() = elem;
    logicalSize++;
}
```

```
    bool empty() const;
    void push_back(const std::string& elem);
    std::string& at(size_t indx); // like vec[] but with error checking


    iterator begin();
    iterator end();
    /*...*/
```

Making `StrVector`'s const-interface

```
class StrVector {
public:
    using iterator = std::string*;
    const size_t kInitialSize = 2;
    /*...*/
    size_t size() const;
    bool empty() const;
    void push_back(const std::string& elem);
    std::string& at(size_t indx); // like vec[] but with error checking


    iterator begin();
    iterator end();
    /*...*/
```

Questions to ask:

1.) Should this function be able available to a const object?
   a.) Can I mark the function const as is (i.e. the function doesn't modify the object)?
   b.) Otherwise, can I make a const version of the function?

Making `StrVector`'s const-interface

```
std::string& StrVector::at(size_t index)  {
    if (index >= size()) {
        throw std::out_of_range("Index out of range in at.");
    }
    return operator[](index); // operator[] =  return *(begin() + index)
}
```

```
cl
pu

    void push_back(const std::string& elem);
    std::string& at(size_t indx); // like vec[] but with error checking


    iterator begin();
    iterator end();
    /*...*/
```

Making `StrVector`'s const-interface

```
std::string& StrVector::at(size_t index)  {
    if (index >= size()) {
        throw std::out_of_range("Index out of range in at.");
    }
    return operator[](index); // operator[] = return *(begin() + index)
}
```

```
cl
pu
    void push_back(const std::string& elem);
    std::string& at(size_t indx); // like vec[] but with error checking

    iterator begin();
    iterator end();
    /*...*/
```

Seems like **at** doesn't modify the vector… can we just mark at const like we did with the other functions?

Making `StrVector`'s const-interface

```
std::string& StrVector::at(size_t index)  {
    if (index >= size()) {
        throw std::out_of_range("Index out of range in at.");
    }
    return operator[](index); // operator[] = return *(begin() + index)
}
```

```
c
pu
    void push_back(const std::string& elem);
    std::string& at(size_t indx); // like vec[] but with error checking


    iterator begin();
    iterator end();
    /*...*/
```

Seems like **at** doesn't modify the vector... can we just mark at const like we did with the other functions? **NO**

Problem: at returns a reference to an element in the vector. That element reference could be modified (thereby modifying the vector).

onst-interface

```
public:
    using iterator = std::string*;
    const size_t kInitialSize = 2;
    /*...*/
    size_t size() const;
    bool empty() const;
    void push_back(const std::string& elem);
    std::string& at(size_t indx); // like vec[] but with error checking


    iterator begin();
    iterator end();
    /*...*/
```

Problem: `at` returns a reference to an element in the vector. That element reference could be modified (thereby modifying the vector).

Example

```
// StrVector my_vec = {"sarah", "haven"};
std::string& elem_ref = my_vec.at(1);
elem_ref = "Now I'm Different";
// my_vec = {"sarah", "Now I'm Different"}
```

onst-interface

```
m);
std::string& at(size_t indx); // like vec[] but with error checking

    iterator begin();
    iterator end();
    /*...*/
```

Making `StrVector`'s const-interface

```
class StrVector {
public:
    using iterator = std::string*;
    const size_t kInitialSize = 2;
    /*...*/
    size_t size() const;
    bool empty() const;
    void push_back(const std::string& elem);
    std::string& at(size_t indx); // like vec[] but with error checking
    const std::string& at(size_t indx) const;

    iterator begin();
    iterator end();
    /*...*/
```

Questions to ask:

1.) Should this function be able available to a const object?
   - a.) Can I mark the function const as is (i.e. the function doesn't modify the object)?
   - b.) Otherwise, can I make a const version of the function?

You might be tempted to do this…

DON'T DO THIS!!!!!!!!!!!!

```
std::string& StrVector::at(size_t index) {
    if (index >= size()) {
        throw std::out_of_range("Index out of range in at.");
    }
    return operator[](index); // operator[] = return *(begin() + index)
}
```

DON'T DO THIS!!!!!!!!!!!!

```
const std::string& StrVector::at(size_t index) const {
    if (index >= size()) {
        throw std::out_of_range("Index out of range in at.");
    }
    return operator[](index); // operator[] = return *(begin() + index)
}
```

DON'T DO THIS!!!!!!!!!!!!

You might be tempted to do this…

DON'T DO THIS!!!!!!!!!!!!

```
std::string& StrVector::at(size_t index) {
    if (index >= size()) {
        throw std::out_of_rang
    }
    return operator[](index);
}
```

This is bad because you are **reimplementing the same logic multiple times** (i.e. copy-pasting). This **makes code harder to maintain** and is more **bug-prone** 😔

DON'T DO THIS!!!!!!!!!!!!

```
const std::string& StrVector::at(size_t index) const {
    if (index >= size()) {
        throw std::out_of_range("Index out of range in at.");
    }
    return operator[](index); // operator[] = return *(begin() + index)
}
```

DON'T DO THIS!!!!!!!!!!!!

Key Idea: Don't reimplement logic unnecessarily

DO THIS!

```
std::string& StrVector::at(size_t index) {
    if (index >= size()) {
        throw std::out_of_range("Index out of range in at.");
    }
    return operator[](index); // operator[] = return *(begin() + index)
}
```

DO THIS!

DO THIS!

```
const std::string& StrVector::at(size_t index) const {
    return static_cast<const
std::string&>(const_cast<StrVector*>(this)->at(index));
}
```

Key Idea: Don’t reimplement logic unnecessarily

DO THIS!

```
std::string& StrVector::at(size_t index) {
    if (index >= size()) {
        throw std::out_of_range("Index out of range in at.");
    }
    return operator[](index); // operator[] = return *(begin() + index)
}
```

DO THIS!

DO THIS!

```
const std::string& StrVector::at(size_t index) const {
    return static_cast<const
std::string&>(const_ca
}
```

This fancy static_cast/const_cast trick allows us to reuse the non-const version to implement the const version! YAY! 💝🙌🎉

Let’s break down this incredibly dense line

```
std::string& StrVector::at(size_t index) {
    if (index >= size()) {
        throw std::out_of_range("Index out of range in at.");
    }
    return operator[](index); // operator[] =  return *(begin() + index)
}
```

1. Casts this (i.e. the current object) from **const StrVector*** to **StrVector*** (i.e. removes the const)

```
const std::string& StrVector::at(size_t index) const {
    return static_cast<const
std::string&>(const_cast<StrVector*>(this)->at(index));
```

```
void printidx2(const StrVector& const_vec) {
   cout << "Item: " << const_vec.at(2) << endl;
}
int main(){ // omitted return 0 for space
   StrVector myVec;
   myVec.push_back("Sarah");
   myVec.push_back("Haven");
   printidx2(myVec);
}
```

ense line

```
in at.");

* (begin() + index)
```

1. Casts this (i.e. the current object) from **const StrVector*** to **StrVector*** (i.e. removes the const)

```
const std::string& StrVector::at(size_t index) const {
   return static_cast<const
std::string&>(const_cast<StrVector*>(this)->at(index));
```

Let’s break down this incredibly dense line

```
std::string& StrVector::at(size_t index) {
    if (index >= size()) {
        throw std::out_of_range("Index out of range in at.");
    }
    return operator[](index); // operator[] = return *(begin() + index)
}
```

2. Calls the non-const version of at on this (remember this is now a StrVector*)

```
const std::string& StrVector::at(size_t index) const {
    return static_cast<const
std::string&>(const_cast<StrVector*>(this)->at(index));
```

Let’s break down this incredibly dense line

```
std::string& StrVector::at(size_t index) {
    if (index >= size()) {
        throw std::out_of_range("Index out of range in at.");
    }
    return operator[](index); // operator[] = return *(begin() + index)
}
```

3. Casts the return of at from **std::string&** to a **const std::string&**

```
const std::string& StrVector::at(size_t index) const {
    return static_cast<const
std::string&>(const_cast<StrVector*>(this)->at(index));
```

Let’s break down this incredibly dense line

```
std::string& StrVector::at(size_t index) {
    if (index >= size()) {
        throw std::out_of_range("Index out o
    }
    return operator[](index); // operator[] =
}
```

In general, this fancy static_cast/const_cast trick allows us to reuse the non-const version to implement the const version! YAY! 💝🙌🎉

The idea is to

1. Cast this (i.e. the current object, which in this situation is const) so it’s pointing to a non-const object
2. Call the non-const version of the function
3. And then cast the non-const return from the function call to a const version

```
const std::string& StrVector::at(size_t index) const {
    return static_cast<const
std::string&>(const_cast<StrVector*>(this)->at(index));
```

Aside: static_cast and const_cast

- **static_cast<new-type>(expression);**
  - Used to convert from one type to another
  - Example: `int my_int = static_cast<int>(3.1);`
  - CANNOT BE USED WHEN conversion would cast away constness

Aside: static_cast and const_cast

- **static_cast<new-type>(expression);**
  - Used to convert from one type to another
  - Example: `int my_int = static_cast<int>(3.1);`
  - CANNOT BE USED WHEN conversion would cast away constness
- **const_cast<new-type>(expression);**
  - const_cast can be used to cast away (remove) constness
  - Allows you to make non-const pointer or reference to const-object

```
const int const_int = 3;
int& my_int = const_cast<int&>(const_int);
```

Making StrVector's const-interface

```
class StrVector {
public:
    using iterator = std::string*;
    const size_t kInitialSize = 2;
    /*...*/
    size_t size() const;
    bool empty() const;
    void push_back(const std::string& elem);
    std::string& at(size_t indx); // like vec[] but with error checking
    const std::string& at(size_t indx) const;

    iterator begin();
    iterator end();
    /*...*/
```

Should `begin()` **and** `end()` **be** `const`**?**

Consider a function with a const `StrVector` param...

```
void printVec(const StrVector& vec){
    cout << "{ ";
    for(auto it = vec.begin(); it != vec.end(); ++it){
        cout << *it << endl;
    }
    cout << " }" << endl;
}
```

These seem like reasonable calls! Let’s mark them const. What could go wrong? :)

Consider a function with a const `StrVector` param...

```
void printVec(const StrVector& vec){
    cout << "{ ";
    for(auto it = vec.begin(); it != vec.end(); ++it){
        *it = "dont mind me modifying a const vector :D";
    }
    cout << " }" << endl;
}
```

This code will compile! begin() and end() don't explicitly change vec, but they give us an iterator that can!

Consider a function with a const `StrVector` param...

```
void printVec(const StrVector& vec){
    cout << "{ ";
    for(auto it = vec.begin(); it != vec.end(); ++it){
        *it = "dont mind me modifying a const vector :D";
    }
    cout << " }" << endl;
}
```

Problem: we need a way to iterate through a const vec just to access it

Solution: const_iterator

```
class StrVector {
public:
    using iterator = std::string*;
    using const_ iterator = const std::string*;
    /*...*/
    size_t size() const;
    bool empty() const;
    /*...*/
    void push_back(const std::string& elem);
    const std::string& at(size_t indx) const;
    iterator begin();
    iterator end();
    const iterator begin()const;
    const_iterator end()const;
    /*...*/
```

Consider a function with a const `StrVector` param...

```
void printVec(const StrVector& vec){
    cout << "{ ";
    for(auto it = vec.begin(); it != vec.end(); ++it){
        *it = "HELLO"; //compile error!
    }
    cout << " }" << cout;
}
```

Fixed! And now we can't set *it equal to something: it will be a compile error!

Abilities of Iterator Permutations

Iterator Type	Increment Iterator?	Change underlying value?
`iterator`	✅	✅
`const_iterator`	✅	❌
`const iterator`	❌	✅
`const const_iterator`	❌	❌

const iterator vs const_iterator: Nitty Gritty

```
using iterator = std::string*;
using const_iterator = const std::string*;

const iterator it_c = vec.begin(); //string * const, const ptr to non-const obj
*it_c = "hi"; //OK! it_c is a const pointer to non-const object
it_c++; //not ok! can't change where a const pointer points!
```

const iterator vs const_iterator: Nitty Gritty

```
using iterator = std::string*;
using const_iterator = const std::string*;

const iterator it_c = vec.begin(); //string * const, const ptr to non-const obj
*it_c = "hi"; //OK! it_c is a const pointer to non-const object
it_c++; //not ok! can't change where a const pointer points!


const_iterator c_it = vec.begin(); //const string*, a non-const ptr to const obj
c_it++; // totally ok! The pointer itself is non-const
*c_it = "hi" // not ok! Can't change underlying const object
cout << *c_it << endl; //allowed! Can always read a const object, just can't change
```

const iterator vs const_iterator: Nitty Gritty

```
using iterator = std::string*;
using const_iterator = const std::string*;

const iterator it_c = vec.begin(); //string * const, const ptr to non-const obj
*it_c = "hi"; //OK! it_c is a const pointer to non-const object
it_c++; //not ok! can’t change where a const pointer points!


const_iterator c_it = vec.begin(); //const string*, a non-const ptr to const obj
c_it++; // totally ok! The pointer itself is non-const
*c_it = "hi" // not ok! Can’t change underlying const object
cout << *c_it << endl; //allowed! Can always read a const object, just can't change

//const string * const, const ptr to const obj
const const_iterator c_it_c = vec.begin();
cout << c_it_c << " points to " << *c_it_c << endl; //only reads are allowed!
c_it_c++; //not ok! can’t change where a const pointer points!
*c_it_c = "hi" // not ok! Can’t change underlying const object
```

realVector.cpp

No more “this is the simplified version of the real thing”… We are writing the real thing (just a little simplified :p)

Recap: Const and Const-correctness

- Use const parameters and variables wherever you can in application code

- Every member function of a class that doesn't change its member variables should be marked `const`

- **`Don't reinvent the wheel! Use our fancy static_cast/const_cast trick to use the non-const version to implement a const version of a function`**

- auto will drop all const and &, so be sure to specify

- Make iterators and const_iterators for all your classes!

Recap: Template classes

- Add `template<class T1, T2..>` before class definition in .h
- Add `template<class T1, T2..>`before all function signatures in .cpp
- When returning nested types (like iterator types), put `typename ClassName<T1, T2..>::member_type` as return type, not just `member_type`
- Templates don’t emit code until instantiated, so `#include` the .cpp file in the .h file, not the other way around!

An unintuitive (at first!) bug

(if time)

The Takeaway

Templates don’t emit code until instantiated, so include the .cpp in the .h instead of the other way around!

A compile error....

```
// vector.h
template <typename T>
class vector<T> {
    T& at(int i);
};
```

```
// vector.cpp
#include "vector.h"
template <typename T>
T& vector<T>::at(int i) {
    // oops
}
```

```
// main.cpp
#include "vector.h"
vector<int> a;
a.at(5);
```

A compile error....

```
// vector.h
template <typename T>
class vector<T> {
    T& at(int i);
};
```

```
// vector.cpp
#include "vector.h"
template <typename T>
T& vector<T>::at(int i) {
    // oops
}
```

```
// main.cpp
#include "vector.h"
vector<int> a;
a.at(5);
```

What the C++ compiler does with **non-template** classes

```
// main.cpp
#include "vectorint.h"
vectorInt a;
a.at(5);
```

1. `g++ -c vectorint.cpp main.cpp`: Compile and create all the code in vectorint.cpp and main.cpp. All the functions in vectorint.h have implementations that have been compiled now, and main can access them because it included vectorint.h

What the C++ compiler does with **non-template** classes

```
// main.cpp
#include "vectorint.h"
vectorInt a;
a.at(5);
```

1. `g++ -c vectorint.cpp main.cpp`: Compile and create all the code in vectorint.cpp and main.cpp. All the functions in vectorint.h have implementations that have been compiled now, and main can access them because it included vectorint.h
2. "Oh look she used vectorInt::at, sure glad I compiled all that code and can access vectorInt::at right now!"

What the C++ compiler does with **template** classes

```
// main.cpp
#include “vector.h”
vector a;
a.at(5);
```

1. `g++ -c vector.cpp main.cpp`: Compile and create all the code in main.cpp. Compile vector.cpp, but since it’s a template, don’t create any code yet.

What the C++ compiler does with **template** classes

```
// main.cpp
#include "vector.h"
vector a;
a.at(5);
```

1. `g++ -c vector.cpp main.cpp`: Compile and create all the code in main.cpp. Compile vector.cpp, but since it's a template, don't create any code yet.

2. "Oh look she made a vector<int>! Better go generate all the code for one of those!"

What the C++ compiler does with **template** classes

```
// main.cpp
#include “vector.h”
vector a;
a.at(5);
```

1. `g++ -c vector.cpp main.cpp`: Compile and create all the code in main.cpp. Compile vector.cpp, but since it’s a template, don’t create any code yet.

2. “Oh look she made a vector<int>! Better go generate all the code for one of those!”

3. “Oh no! All I have access to is vector.h! There’s no implementation for the interface in that file! And I can’t go looking for vector<int>.cpp!”

The fix...

```
// vector.h
template <typename T>
class vector<T> {
    T& at(int i);
};
```

```
// vector.cpp
#include "vector.h"
template <typename T>
T& vector<T>::at(int i) {
    // oops
}
```

```
// main.cpp
#include "vector.h"
vector<int> a;
a.at(5);
```

Include vector.cpp in vector.h!

```
// vector.h
#include "vector.cpp"
template <typename T>
class vector<T> {
    T& at(int i);
};
```

```
// vector.cpp

template <typename T>
T& vector<T>::at(int i) {
    // oops
}
```

```
// main.cpp
#include "vector.h"
vector<int> a;
a.at(5);
```

What the C++ compiler does with template classes

```
// main.cpp
#include "vector.h"
vector<int> a;
a.at(5);
```

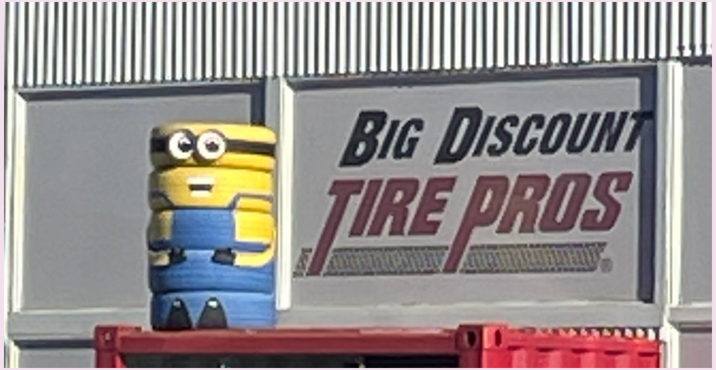
1. "Oh look she included vector.h! That's a template, **I'll wait to link the implementation until she instantiates a specific kind of vector**"
2. "Oh look she made a vector<int>! Better go generate all the code for one of those!"
3. "vector.h includes all the code in vector.cpp, which tells me how to create a vector<int>::at function :)"

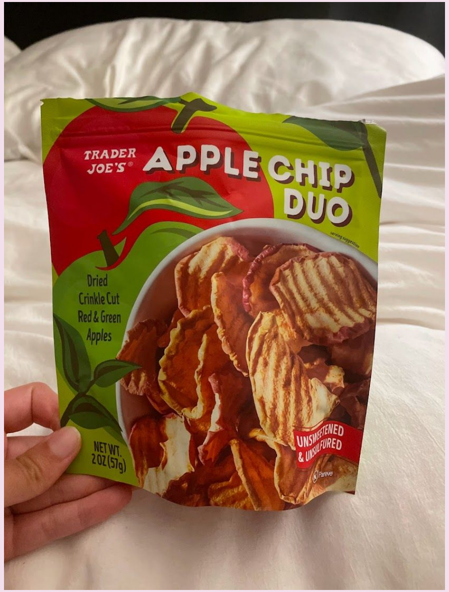
The Takeaway

Templates don’t emit code until instantiated, so include the .cpp in the .h instead of the other way around!

Practice with Template Classes!

- Implement Vector yourself!
- We got you started with some basic functionality but you can write many more methods including: insert, erase, pop_back. See documentation for notes about functionality





Thanks for coming!

Next time: Template Functions