Template Classes + Const Correctness

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

Today

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

Definition

Class: A programmer-defined custom type. An abstraction of an object or data type.

Turning `Student` into a class: basic components

//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 **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

Code: StrVector

Today

- ~~Finish StrVector~~
- **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.

Template Classes You’ve Used

- Vectors!

```
vector<int> numVec; vector<string> strVec;
```

Template Classes You've Used

- Vectors!

```
vector<int> numVec; vector<string> strVec;
```

- Maps!

```
map<int, string> int2Str; map<int, int> int2Int;
```

Template Classes You’ve Used

- Vectors!

```
vector<int> numVec; vector<string> strVec;
```

- Maps!

```
map<int, string> int2Str; map<int, int> int2Int;
```

- Sets!

```
set<int> someNums; set<Student> someStudents;
```

Template Classes You've Used

- Vectors!

```
vector<int> numVec; vector<string> strVec;
```

- Maps!

```
map<int, string> int2Str; map<int, int> int2Int;
```

- Sets!

```
set<int> someNums; set<Student> someStudents;
```

Pretty much all containers!

Fundamental Theorem of Software Engineering: Any problem can be solved by adding enough layers of indirection.

Writing a template: Syntax

```
//Example: Structs
template<typename First, typename Second> struct MyPair {
    First first;
    Second second;
};
```

```
//Exactly Functionally the same!
template<typename One, typename Two> struct MyPair {
    One first;
    Two second;
};
```

Writing a Template Class: Syntax

//mypair.h

```
template<class First, class Second> class MyPair {
    public:
        /*...*/
    private:
        First first;
        Second second;
};
```

Writing a Template Class: Syntax

```
//mypair.h
template<class First, class Second> class MyPair {
    public:
        /*...*/
    private:
        First first;
        Second second;
};
```

Writing a Template Class: Syntax

//mypair.h

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

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

Writing a Template Class: Syntax

```
//mypair.h
template<class First, class Second> class MyPair {
    public:
        First getFirst();
        Second getSecond();

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

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;
}
//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;
}
```

Implementing a Template Class: Syntax

//mypair.cpp

```
#include "mypair.h"

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

template<class Second, typename First>
Second MyPair<First, Second>::getSecond(){
    return second;
}
```

realVector.cpp

No more “this is the simplified version of the real thing”... We are writing the real thing!

Member Types

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

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

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

Member Types: Syntax

```
//vector.h
template<typename T> class vector {
    using iterator = …  // something internal

    private:
    iterator front;
}
```

Member Types: Syntax

```
//vector.h
template<typename T> class vector {
    using iterator = …  // something internal

    private:
    iterator front;
}
```

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

Member Types: Syntax

```
//vector.h
template<typename T> class vector {
    using iterator = …  // something internal

    private:
    iterator front;
}
```

```
//vector.cpp
template <typename T>
iterator vector<T>::insert(iterator pos, int value) {...}
//iterator is a nested type in namespace vector<T>::
```

Member Types: Syntax

```
//vector.h
template<typename T> class vector {
    using iterator = …  // something internal

    private:
    iterator front;
}
```

```
//vector.cpp
template <typename T>
typename vector<T>::iterator vector<T>::insert(iterator pos, int
value) {...}
```

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.**

One final compile error....

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

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

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

```
g++ -c vector.cpp main.cpp
g++ vector.o main.o -o output
```

One final compile error….

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

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

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

```
g++ -c vector.cpp main.cpp
g++ vector.o main.o -o output
```

Templates don’t emit code until instantiated

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.
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...

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

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

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

```
g++ -c vector.cpp main.cpp
g++ vector.o main.o -o output
```

Include vector.cpp in vector.h!

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

```
// vector.cpp

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

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

```
g++ -c vector.cpp main.cpp
g++ vector.o main.o -o output
```

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. "vector.h includes all the code in vector.cpp, which tells me how to create a vector<int>::at function :)"

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

realVector.cpp

No more “this is the simplified version of the real thing”... We are writing the real thing!

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!

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 to an non-const variable

vec.push_back(4);
c_vec.push_back(9);
ref.push_back(5);
c_ref.push_back(6);
```

`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(4);     // OKAY
c_vec.push_back(9);
ref.push_back(5);
c_ref.push_back(6);
```

`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(4);     // OKAY
c_vec.push_back(9);   // BAD - const
ref.push_back(5);
c_ref.push_back(6);
```

`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(4);     // OKAY
c_vec.push_back(9);   // BAD - const
ref.push_back(5);    // OKAY
c_ref.push_back(6);
```

`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(4);     // OKAY
c_vec.push_back(9);   // BAD - const
ref.push_back(5);    // OKAY
c_ref.push_back(6); // BAD - const
```

Can’t declare non-const reference to const variable!

```
const std::vector<int> c_vec{7, 8};  // a const variable

// BAD - can't declare non-const ref to const vector
std::vector<int>& bad_ref = c_vec;
```

Can’t declare non-const reference to const variable!

```
const std::vector<int> c_vec{7, 8};  // a const variable

// fixed
const std::vector<int>& bad_ref = c_vec;
```

Can’t declare non-const reference to const variable!

```
const std::vector<int> c_vec{7, 8};  // a const variable

// fixed
const std::vector<int>& bad_ref = c_vec;

// BAD - Can't declare a non-const reference as equal
// to a const reference!
std::vector<int>& ref = c_ref;
```

`const` & subtleties with auto

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};

std::vector<int>& ref = vec;
const std::vector<int>& c_ref = vec;

auto copy = c_ref;              // a non-const copy
const auto copy = c_ref;        // a const copy
auto& a_ref = ref;              // a non-const reference
const auto& c_aref = ref;       // a const reference
```

Why const?

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;
}
```

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;
}
```

Why const? Overly ambitious functions in application code

```
long int countPopulation(Planet& p) {
 // Hats are the cornerstone of modern society
 addLittleHat(p);
 return p.count_population();
}
```

Why const? Overly ambitious functions in application code

```
long int countPopulation(Planet& p) {
 // Hats are the cornerstone of modern society
 addLittleHat(p);
 // Guaranteed no more population growth, all future calls will be faster
 sterilize(p);
 return p.count_population();
}
```

Why const? Overly ambitious functions in application code

```
long int countPopulation(Planet& p) {
 // Hats are the cornerstone of modern society
 addLittleHat(p);
 // Guaranteed no more population growth, all future calls will be faster
 sterilize(p);
 // Optimization: destroy planet
 // This makes population counting very fast
 deathStar(p);
 return 0;
}
```

Why const? Overly ambitious functions in application code

```
long int countPopulation(const Planet& p) {
 // Hats are the cornerstone of modern society
 addLittleHat(p); //compile error
 // Guaranteed no more population growth, all future calls will be faster
 sterilize(p);  //compile error
 // Optimization: destroy planet
 // This makes population counting very fast
 deathStar(p); //compile error
 return 0;
}
```

How does this work?

```
long int countPopulation(const Planet& p) {
 // Hats are the cornerstone of modern society
 addLittleHat(p); //compile error
 ...
 }
void addLittleHat(Planet& p) {
 p.add(👒);
}
```

p is a **const reference** here

p is a **(non const) reference** here

Calling addLittleHat on p is like setting a non const variable equal to a const one, it's not allowed!

Code: make RealVector const correct

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." ;
}
//compile error!
```

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("Age cannot be
negative!");
}
```

//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.

Making `RealVector`'s const-interface

```
class StrVector {
public:
    using iterator = std::string*;
    const size_t kInitialSize = 2;
    /*...*/
    size_t size();
    bool empty();
    std::string& at(size_t indx);
    void insert(size_t pos, const std::string& elem);
    void push_back(const std::string& elem);

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

Making `RealVector`'s const-interface

```
class StrVector {
public:
    using iterator = std::string*;
    const size_t kInitialSize = 2;
    /*...*/
    size_t size() const;
    bool empty() const;
    std::string& at(size_t indx);
    void insert(size_t pos, const std::string& elem);
    void push_back(const std::string& elem);

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

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

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

```
void printVec(const RealVector& 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 `RealVector` param...

```
void printVec(const RealVector& 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 `RealVector` param...

```
void printVec(const RealVector& 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: `cbegin()` and `cend()`

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

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

```
void printVec(const RealVector& vec){
    cout << "{ ";
    for(auto it = vec.cbegin(); it != vec.cend(); ++it){
        cout << *it << cout;
    }
    cout << " }" << cout;
}
```

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

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! cant change where a const pointer points!


const_iterator c_it = vec.cbegin(); //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.cbegin();
cout << c_it_c << " points to " << *c_it_c << endl; //only reads are allowed!
```

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`
- auto will drop all const and &, so be sure to specify
- Make iterators and const_iterators for all your classes!
  - **`const iterator`** = cannot increment the iterator, can dereference and change underlying value
  - **`const_iterator`** = can increment the iterator, cannot dereference and change underlying value
  - **`const const_iterator`** = cannot increment iterator, cannot dereference and change underlying value

Recap: Template classes

- Add `template<typename T1, typename T2..>` before class definition in .h
- Add `template<typename T1, typename T2..>` before all function signature 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!