

Classes

...

How to make your own custom types!

Today



- **Recap: Containers + Iterators**
- Classes Introduction
- Template Classes (intro)

Containers...contain things!

`std::vector`

- Use for almost everything
- Backed by an array! (more later)
- We will implement one today!

Containers...contain things!

std::vector - use for almost everything

std::deque

- use when you need fast insertion to front AND back
- implemented using arrays of arrays (or sometimes linked lists)

Containers...contain things!

std::vector - use for almost everything

std::deque - when you need fast insertion to front AND back

std::stack and **std::queue**

- Last In First Out (LIFO) and First In First Out (FIFO)
- Built on top on dequeues!

Containers...contain things!

std::vector - use for almost everything

std::deque - when you need fast insertion to front AND back

std::stack and **std::queue** - LIFO and FIFO

std::map and **std::set**

- Fast insertion, deletion and access!
- Sets don't guarantee any order, but are ordered
- Maps are basically sets of `std::pairs`!

Containers...contain things!

std::vector - use for almost everything

std::deque - when you need fast insertion to front AND back

std::stack and **std::queue** - LIFO and FIFO

std::map and **std::set** - Fast everything, no indexing

std::unordered_map and **std::unordered_set**

- Even faster everything, no indexing, your keys need to be hashable!

Questions?

Containers are all classes defined in the STL!

...

Today, we will be learning about making our OWN classes!

Iterators

```
std::vector<int> myVec{1, 2, 3, 4};  
for(auto it = myVec.begin(); it != myVec.end(); ++it){  
    cout << *it << endl;  
}
```

Iterators

```
std::vector<int> myVec{1, 2, 3, 4};  
for(auto it = myVec.begin(); it != myVec.end(); ++iter){  
    //it has type std::vector<int>::iterator  
    cout << *it << endl;  
}
```

```
std::set<int> mySet{1, 2, 3, 4};  
for(auto it = mySet.begin(); it != mySet.end(); ++iter){  
    //it has type std::set<int>::iterator  
    cout << *it << endl;  
}
```

**Iterators are (basically) pointers! More on that
later**

Questions?

Today



- ~~Recap: Containers + Iterators~~
- **Classes Introduction**
- Template Classes (intro)

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

Definition

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

But don't structs do that?

```
struct Student {  
    string name; // these are called fields  
    string state; // separate these by semicolons  
    int age;  
};
```

```
Student s = {"Frankie", "MN", 21};
```

Issues with structs

- Public access to all internal state data by default

```
Student s = {"Frankie", "MN", 21};
```

```
s.age = -5;
```

```
//should guard against nonsensical values
```

Issues with structs

- Public access to all internal state data by default
- Users of struct need to explicitly initialize each data member.

```
Student s;  
cout << s.name << endl; //s.name is garbage  
s.name = "Frankie";  
cout << s.name << endl; //now we're good!
```

“A struct simply feels like an open pile of bits with very little in the way of encapsulation or functionality. A class feels like a living and responsible member of society with intelligent services, a strong encapsulation barrier, and a well defined interface.”

- Bjarne Stroustrup

Classes provide their users with a **public interface** and separate this from a **private implementation**

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;  
};
```

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

//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

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) {  
}
```


Recall: namespaces

- Put code into logical groups, to avoid name clashes
- Each class has its own namespace
- Syntax for calling/using something in a namespace:

```
namespace_name::name
```

Function definitions with namespaces!

- `namespace_name::name` in a function prototype means “this is the implementation for an interface function in `namespace_name`”
- Inside the `{ ... }` the private member variables for `namespace_name` will be in scope!

```
std::string Student::getName() { ... }
```

//student.cpp

```
#include student.h

std::string Student::getName() {
    return name; //we can access name here!
}

void Student::setName(std::string name) {

}

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;
};
```

//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`”

```
void Student::setName(string name) {  
    name = name; //huh?  
}
```

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 naming conflicts!

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

//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) {

}
```

//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;
};
```


Constructors

- Define how the member variables of an object is initialized
- What gets called when you first create a Student object

```
//student.cpp
```

```
#include student.h
```

```
Student::Student() {
```

```
    age = 0;
```

```
    name = "";
```

```
    state = "";
```

```
}
```

Constructors

- Define how the member variables of an object is initialized
- What gets called when you first create a Student object
- Overloadable!

//student.cpp

```
#include student.h
```

```
Student::Student() {...}
```

```
Student::Student(string name, int age, string state){
```

```
    this->name = name;
```

```
    this->age = age;
```

```
    this->state = state;
```

```
}
```

Putting it all together: Using your shiny new class!

```
//main.cpp
```

```
#include student.h
```

```
int main() {
```

```
    Student frankie;
```

```
    frankie.setName("Frankie");
```

```
    frankie.setAge(21);
```

```
    frankie.setState("MN");
```

```
    cout << frankie.getName() << " is from " << frankie.getState() <<
```

```
endl;
```

```
}
```

Putting it all together: Using your shiny new class!

//main.cpp

```
#include student.h
```

```
int main() {
```

```
    Student frankie;
```

```
    frankie.setName("Frankie");
```

```
    frankie.setAge(21);
```

```
    frankie.setState("MN");
```

```
    cout << frankie.getName() << " is from " << frankie.getState();
```

```
    Student sathya("Sathya", 20, "New Jersey");
```

```
    cout << sathya.getName() << " is from " << sathya.getState();
```

```
}
```

Code: `vectorint.cpp`

...

Let's build a more complicated class!

One last thing... Arrays

- Arrays are a primitive type! They are the building blocks of all containers
- Think of them as lists of objects of fixed size that you can index into
- Think of them as the struct version of vectors. You should not be using them in application code! Vectors are the STL interface for arrays!

```
//int * is the type of an int array variable
```

```
int *my_int_array;
```

```
//this is how you initialize an array
```

```
my_int_array = new int[10];
```

```
//this is how you index into an array
```

```
int one_element = my_int_array[0];
```

One last thing... Arrays

*//int * is the type of an int array variable*

```
int *my_int_array;
```

//my_int_array is a pointer!

//this is how you initialize an array

```
my_int_array = new int[10];
```

+--+--+--+--+--+--+--+--+--+

//my_int_array -> | | | | | | | | | |

+--+--+--+--+--+--+--+--+--+

//this is how you index into an array

```
int one_element = my_int_array[0];
```

Destructors

- Arrays are memory **WE** allocate, so we need to give instructions for when to deallocate that memory!
- When we are done using our array, we need to delete [] it!

```
//int * is the type of an array variable
```

```
int *my_int_array;
```

```
//this is how you initialize an array
```

```
my_int_array = new int[10];
```

```
//this is how you index into an array
```

```
int one_element = my_int_array[0];
```

```
delete [] my_int_array;
```


Destructors

- deleting (almost) always happens in the **destructor** of a class!
- The destructor is defined using `Class_name::~~Class_name()`
- No one ever explicitly calls it! Its called when `Class_name` object go out of scope!
- Just like all member functions, declare it in the .h and implement in the .cpp!

Code: strvector.cpp

...

For real!

Today



- ~~— Recap: Containers + Iterators~~
- ~~— Classes Introduction~~
- **Template Classes (intro)**

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

The problem with IntVector

- Vectors should be able to contain any data type!

The problem with IntVector

- Vectors should be able to contain any data type!

Solution? Create StringVector, DoubleVector, BoolVector etc..

The problem with IntVector

- Vectors should be able to contain any data type!

Solution? Create StringVector, DoubleVector, BoolVector etc..

- What if we want to make a vector of Students?
 - 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 IntVector

- Vectors should be able to contain any data type!

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

- What if we want to make a vector of `Students`?
 - 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!

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!

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

Writing a Template Class: Syntax

//mypair.h

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


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;  
};
```

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;  
};
```

Use generic typename 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<typename First, typename Second>
```

```
First MyPair::getFirst() {
```

```
    return first;
```

```
}
```

Implementing a Template Class: Syntax

```
//mypair.cpp
```

```
#include "mypair.h"
```

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

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

One final compile error....

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

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

```
// 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);
```

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
```


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...

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

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

```
// 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);
```

Include vector.cpp in vector.h!

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

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

```
// vector.cpp
```

```
template <typename T>
void 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 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 :)”

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

Next time: vector.cpp

...

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