Types and Structs

Types make things better...and sometimes harder...but still better >:(

😷 masks required
Recap
C++: Basic Syntax + the STL

Basic syntax
- Semicolons at EOL
- Primitive types (ints, doubles etc)
- Basic grammar rules

The STL
- Tons of general functionality
- Built in classes like maps, sets, vectors
- Accessed through the namespace std:::
Standard C++: Basic Syntax + std library

The STL

- Tons of general functionality
- Built in classes like maps, sets, vectors
- Accessed through the namespace std::
- Extremely powerful and well-maintained
Namespaces

- MANY things are in the `std::` namespace
  - e.g. `std::cout`, `std::cin`, `std::lower_bound`
- CS 106B always uses the `using namespace std;` declaration, which automatically adds `std::` for you
- We won’t (most of the time)
  - it’s not good style!
Today

- Types
- Intro to structs
- Sneak peek at streams!
C++ Fundamental Types

```cpp
int val = 5; // 32 bits
char ch = 'F'; // 8 bits (usually)
float decimalVal1 = 5.0; // 32 bits (usually)
double decimalVal2 = 5.0; // 64 bits (usually)
bool bVal = true; // 1 bit
```
```cpp
#include <string>

int val = 5;  // 32 bits
char ch = 'F';  // 8 bits (usually)
float decimalVal1 = 5.0;  // 32 bits (usually)
double decimalVal2 = 5.0;  // 64 bits (usually)
bool bVal = true;  // 1 bit

std::string str = "Sarah";
```
Fill in the types!

```cpp
a = "test";
b = 3.2 * 5 - 1;
c = 5 / 2;

d(int foo) { return foo / 2; }
e(double foo) { return foo / 2; }
f(double foo) { return int(foo / 2); }
g(double c) {
    std::cout << c << std::endl;
}
```
string a = "test";
double b = 3.2 * 5 - 1;
int c = 5 / 2;       // int/int → int, what’s the value?

_____ d(int foo) { return foo / 2; }
_____ e(double foo) { return foo / 2; }
_____ f(double foo) { return int(foo / 2); }

_____ g(double c) {
    std::cout << c << std::endl;
}
string a = "test";

double b = 3.2 * 5 - 1;

int c = 5 / 2;          // int/int → int, what’s the value?

int d(int foo) { return foo / 2; }

double e(double foo) { return foo / 2; }

int f(double foo) { return int(foo / 2); }

____ g(double c) {
    std::cout << c << std::endl;
}

Fill in the types!

```cpp
string a = "test";
double b = 3.2 * 5 - 1;
int c = 5 / 2; // int/int → int, what's the value?

int d(int foo) { return foo / 2; }
double e(double foo) { return foo / 2; }
int f(double foo) { return int(foo / 2); }

void g(double c) {
    std::cout << c << std::endl;
}
```
C++ is a statically typed language
**statically typed**: everything with a name (variables, functions, etc) is given a type before runtime
**Definition**

**dynamically typed**: everything with a name (variables, functions, etc) is given a type at runtime based on the thing’s current value.
Compiled vs Interpreted

Main Difference: When is source code translated?

Source Code: Original code, usually typed by a human into a computer (like C++ or Python)

Translated: Converting source code into something a computer can understand (i.e. machine code)
Compiled vs Interpreted: When is source code translated?

Dynamically typed, interpreted
- Types checked on the fly, during execution, line by line
- Example: Python

Statically typed, compiled
- Types before program runs during compilation
- Example: C++

Runtime: Period when program is executing commands (after compilation, if compiled)
C++ Types in Action

```cpp
int a = 3;
string b = "test";

char func(string c) {
    // do something
}

b = "test two";

func(b);

// don’t need to declare type after initialization
```
Dynamic vs Static typing: Python vs C++

**Python**

```python
a = 3
b = "test"
def func(c):
    # do something
```

**C++**

```cpp
int a = 3;
string b = "test";
char func(string c) {
    // do something
}
```
Dynamic vs Static typing: Python vs C++

Python

```python
val = 5;
bVal = true;
str = "hi";
```

C++

```cpp
int val = 5;
bool bVal = true;
string str = "hi";
```
Dynamic vs Static typing: Python vs C++

**Python**

```
val = 5
bVal = true
str = "hi"
val = "hi"
str = 100
```

**C++**

```
int val = 5;
bool bVal = true;
string str = "hi";
```
Dynamic vs Static typing: Python vs C++

Python

val = 5
bVal = true
str = "hi"
val = "hi"
str = 100

C++

int val = 5;
bool bVal = true;
string str = "hi";
val = "hi";
str = 100;

ERROR!
Dynamic vs Static typing: Python vs C++

**Python**

```python
def div_3(x):
    return x / 3

div_3("hello")
```

**C++**

```c++
int div_3(int x) {
    return x / 3;
}

div_3("hello")
```
Dynamic vs Static typing: Python vs C++

Python

```python
def div_3(x):
    return x / 3

div_3("hello")
```

//CRASH during runtime, can’t divide a string

C++

```cpp
int div_3(int x) {
    return x / 3;
}

div_3("hello")
```
Dynamic vs Static typing: Python vs C++

Python

```python
def div_3(x):
    return x / 3

div_3("hello")

//CRASH during runtime, can’t divide a string
```

C++

```c++
int div_3(int x) {
    return x / 3;
}

div_3("hello")

//Compile error: this code will never run
```
Dynamic vs Static typing: Python vs C++

Python

def mul_3(x):
    return x * 3

mul_3("10")

C++

int mul_3(int x) {
    return x * 3;
}

mul_3("10");
Dynamic vs Static typing: Python vs C++

Python

def mul_3(x):
    return x * 3

mul_3("10")

//returns “101010”

C++

int mul_3(int x) {
    return x * 3;
}

mul_3("10");

//Compile error: “10” is a string! This code won’t run
Dynamic vs Static typing: Python vs C++

Python

def add_3(x):
    return x + 3

add_3("10")

//returns “103”

C++

int add_3(int x) {
    return x + 3;
}

add_3("10");

//Compile error: “10” is a string! This code won’t run
static typing helps us to prevent errors before our code runs
Static Types + Functions

Python

```python
def div_3(x)
div_3: __ -> ??
```

C++

```cpp
int div_3(int x)
div_3: int -> int
```
C++ to Python, probably
Static Types + Functions

What are the “types” of the following functions?

```c
int add(int a, int b);
    int, int -> int

string echo(string phrase);

string helloworld();

double divide(int a, int b);
```
What are the “types” of the following functions?

```plaintext
int add(int a, int b);
    int, int -> int
string echo(string phrase);
    string -> string
string helloworld();

double divide(int a, int b);
```
What are the “types” of the following functions?

- `int add(int a, int b);`
  - int, int -> int
- `string echo(string phrase);`
  - string -> string
- `string helloworld();`
  - void -> string
- `double divide(int a, int b);`
What are the “types” of the following functions?

```c
int add(int a, int b);
    int, int -> int
string echo(string phrase);
    string -> string
string helloworld();
    void -> string
double divide(int a, int b);
    int, int -> double
```
Questions?
Overloading

- What if we want two versions of a function for two different types?
- Example: int division vs double division
Overloading
Define two functions with the same name but different types

```cpp
int half(int x) {
    std::cout << "1" << endl;      // (1)
    return x / 2;
}

double half(double x) {
    cout << "2" << endl;       // (2)
    return x / 2;
}

half(3)          // uses version (1), returns ?
half(3.0)        // uses version (2), returns ?
```
Overloading
Define two functions with the same name but different types

```cpp
int half(int x) {
    std::cout << "1" << endl; // (1)
    return x / 2;
}

double half(double x) {
    cout << "2" << endl; // (2)
    return x / 2;
}

half(3) // uses version (1), returns 1
half(3.0) // uses version (2), returns 1.5
```
Overloading

Define two functions with the same name but different types

```c
int half(int x, int divisor = 2) {   // (1)
    return x / divisor;
}
double half(double x) {    // (2)
    return x / 2;
}

half(4) // uses version ??, returns ??
half(3, 3) // uses version ??, returns ??
half(3.0) // uses version ??, returns ??
```
Overloading
Define two functions with the same name but different types

```cpp
int half(int x, int divisor = 2) {       // (1)
    return x / divisor;
}

double half(double x) {       // (2)
    return x / 2;
}

half(4) // uses version (1), returns 2
half(3, 3) // uses version (1), returns 1

half(3.0) // uses version (2), returns 1.5
```
Questions?
Today

- Types
  - Intro to structs
  - Sneak peek at streams!
struct: a group of named variables each with their own type. A way to bundle different types together.
struct Student {
    string name;  // these are called fields
    string state; // separate these by semicolons
    int age;
};

Student s;
s.name = "Sarah";
s.state = "CA";
s.age = 21;  // use . to access fields
Use structs to pass around grouped information

```cpp
Student s;
s.name = "Sarah";
s.state = "CA";
s.age = 21; // use . to access fields

void printStudentInfo(Student student) {
    cout << s.name << " from " << s.state;
    cout << " (" << s.age ")
```
Use structs to return grouped information

```cpp
Student randomStudentFrom(string state) {
    Student s;
    s.name = "Sarah"; // random = always Sarah
    s.state = state;
    s.age = randint(0, 100);
    return s;
}

Student foundStudent = randomStudentFrom("CA");
cout << foundStudent.name << endl; // Sarah
```
Abbreviated Syntax to Initialize a struct

Student s;
s.name = "Sarah";
s.state = "CA";
s.age = 21;

//is the same as ...
Abbreviated Syntax to Initialize a struct

```c
Student s;
s.name = "Sarah";
s.state = "CA";
s.age = 21;

// is the same as ...
Student s = {"Sarah", "CA", 21};
```
Questions?
`std::pair`: An STL built-in struct with two fields *of any type*.
**std::pair**

- **std::pair** is a **template**: You specify the types of the fields inside <> for each pair object you make.
- The fields in **std::pairs** are named **first** and **second**.

```cpp
std::pair<int, string> numSuffix = {1,"st"};
cout << numSuffix.first << numSuffix.second;  //prints 1st
```
**std::pair**

- **std::pair** is a **template**: You specify the types of the fields inside <> for each pair object you make.
- The fields in **std::pair**s are named **first** and **second**.

```cpp
struct Pair {
    fill_in_type first;
    fill_in_type second;
};
```
Use `std::pair` to return success + result

```cpp
std::pair<bool, Student> lookupStudent(string name) {
    Student blank;
    if (found(name)) return std::make_pair(false, blank);
    Student result = getStudentWithName(name);
    return std::make_pair(true, result);
}

std::pair<bool, Student> output = lookupStudent("Julie");
```
Use `std::pair` to return success + result

```cpp
std::pair<bool, Student> lookupStudent(string name) {
    Student blank;
    if (notFound(name)) return std::make_pair(false, blank);
    Student result = getStudentWithName(name);
    return std::make_pair(true, result);
}
std::pair<bool, Student> output = lookupStudent("Julie");
```

To avoid specifying the types of a pair, use `std::make_pair(field1, field2)`
Questions?
Aside: Type Deduction with auto
**auto**: Keyword used in lieu of type when declaring a variable, tells the compiler to deduce the type.
Type Deduction using `auto`

```cpp
// What types are these?
auto a = 3;
auto b = 4.3;
auto c = 'X';
auto d = "Hello";
auto e = std::make_pair(3, "Hello");
```

📝 `auto` does not mean that the variable doesn’t have a type. It means that the type is **deduced** by the compiler.
Type Deduction using `auto`

```
// What types are these?
auto a = 3;
auto b = 4.3;
auto c = 'X';
auto d = "Hello";
auto e = std::make_pair(3, "Hello");
```

**Answers:** int, double, char, char* (a C string), std::pair<int, char*>.

💡 `auto` does not mean that the variable doesn’t have a type. It means that the type is **deduced** by the compiler.
!! `auto` does not mean that the variable doesn’t have a type.

It means that the type is deduced by the compiler.
Code Demo!
quadratic.cpp
a general quadratic equation can always be written:

$$ax^2 + bx + c = 0$$

the solutions to a general quadratic equation are:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Radical

If Radical < 0, no real roots.
Recap

- Everything with a name in your program has a **type**
- **Strong type systems** prevent errors before your code runs!
- **Structs** are a way to bundle a bunch of variables of many types
- **std::pair** is a type of struct that had been defined for you and is in the STL
- So you access it through the **std::namespace** (std::pair)
- **auto** is a keyword that tells the compiler to deduce the type of a variable, it should be used when the type is obvious or very cumbersome to write out
Today

- Types
- Intro to structs
- Sneak peek at streams!
stream: an abstraction for input/output. Streams convert between data and the string representation of data.
A stream you’ve used: cout

```cpp
std::cout << 5 << std::endl; // prints 5
// use a stream to print any primitive type!
std::cout << "Sarah" << std::endl;
```
A stream you’ve used: `cout`

```cpp
std::cout << 5 << std::endl; // prints 5
// use a stream to print any primitive type!
std::cout << "Sarah" << std::endl;
// Mix types!
std::cout << "Sarah is " << 21 << std::endl;
```
std::cout << 5 << std::endl;  // prints 5
// use a stream to print any primitive type!
std::cout << "Sarah" << std::endl;
// Mix types!
std::cout << "Sarah is " << 21 << std::endl;
// structs?
Student s ={"Sarah", "CA", 21};
std::cout << s << std::endl;
A stream you’ve used: `cout`

```cpp
std::cout << 5 << std::endl;  // prints 5
// use a stream to print any primitive type!
std::cout << "Sarah" << std::endl;
// Mix types!
std::cout << "Sarah is " << 21 << std::endl;
// structs?
Student s = {"Sarah", "CA", 21};
std::cout << s << std::endl;
```
A stream you’ve used: `cout`

```cpp
std::cout << 5 << std::endl; // prints 5

// use a stream to print any primitive type!

std::cout << "Sarah" << std::endl;

// Mix types!

std::cout << "Sarah is " << 21 << std::endl;

// structs?

Student s = {"Sarah", "CA", 21};
std::cout << s.name << s.age << std::endl;
```
A stream you’ve used: `cout`

```cpp
std::cout << 5 << std::endl; // prints 5
// use a stream to print any primitive type!
std::cout << "Sarah" << std::endl;
// Mix types!
std::cout << "Sarah is " << 21 << std::endl;
// Any primitive type + most from the STL work!
// For other types, you will have to write the
<< operator yourself!
```