http://web.stanford.edu/class/cs106l/

Types and Structs

The blessings (and curses) of a typed language and
how to use it properly!

Haven Whitney and Fabio Ibanez

Fall 2023

Logistics Recap

As a reminder, all course materials can be found on our class website at cs106l.stanford.edu!

- If you missed the first lecture, definitely look over the slides for the Welcome lecture because we covered the important logistics there.

We have an anonymous feedback form that is open all quarter! Feel free to communicate with us through that as well!

Types and Structs

http://web.stanford.edu/class/cs106l/

Agenda

01. C++ Basics

Comparing to other languages you might know

02. Types

Working with a statically typed language

03. Structs

...and pairs and structured binding

Agenda

01. **C++ Basics**
Comparing to other languages you might know

02. **Types**
Working with a statically typed language

03. **Structs**
...and pairs and structured binding

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::

Types and Structs

http://web.stanford.edu/class/cs106l/

C++: Basic Syntax + the STL

Basic syntax

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

s,

This is some C++ code...

```
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

This is some C++ code...

```
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

To import additional functionality, we use #include

This is some C++ code...

```
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

All lines/statements end in a semicolon.

This is some C++ code...

```
#include <io

int main() {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

Function declarations need the type of the return, the name of the function, and any parameters (inside the parentheses).

This is some C++ code…

```
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

The code of a function lives inside a block marked by curly brackets { }.

Types and Structs

http://web.stanford.edu/class/cs106l/

C++: Basic Syntax + the STL

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!

STL Naming Conventions

STL = Standard Template Library

- It contains TONS of functionality (algorithms, containers, functions, iterators), some of which we will explore in this class.

`STD::` = the STL namespace

So to access elements from the **STL** use **`std::`** !

Questions?

Types and Structs

http://web.stanford.edu/class/cs106l/

Agenda

01. **C++ Basics**
Comparing to other languages you might know

02. **Types**
Working with a statically typed language

03. **Structs**
...and pairs and structured binding

Types and Structs

http://web.stanford.edu/class/cs106l/

Fundamental Types

```
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
```

Fundamental Types++

```
#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 = "Haven";
```

Types and Structs

http://web.stanford.edu/class/cs106l/

Fill in the blanks!

```
______ 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;
}
```

Types and Structs

http://web.stanford.edu/class/cs106l/

Fill in the blanks!

```
std::string 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;
}
```

Types and Structs

http://web.stanford.edu/class/cs106l/

Fill in the blanks!

```
std::string a = "test";
double 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;
}
```

Types and Structs

http://web.stanford.edu/class/cs106l/

Fill in the blanks!

```
std::string a = "test";
double b = 3.2 * 5 - 1;
int 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;
}
```

Fill in the blanks!

```
std::string a = “test”;
double b = 3.2 * 5 - 1;
int c = 5 / 2;

int 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;
}
```

Types and Structs

http://web.stanford.edu/class/cs106l/

Fill in the blanks!

```
std::string a = "test";
double b = 3.2 * 5 - 1;
int c = 5 / 2;

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

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

Fill in the blanks!

```
std::string a = "test";
double b = 3.2 * 5 - 1;
int c = 5 / 2;

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;
}
```

Types and Structs

http://web.stanford.edu/class/cs106l/

Fill in the blanks!

```
std::string a = "test";
double b = 3.2 * 5 - 1;
int c = 5 / 2;

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.**

C++ is a statically typed language!

Statically typed: everything with a name (variables, functions, etc) is given a type **before runtime.**

A language like Python is **dynamically typed:** everything with a name (variables, functions, etc) is given a type at runtime based on the thing's **current value**

C++ is a statically typed language!

Statically typed: everything with a name (variables, functions, etc) is given a type **before runtime.**

A language like Python is **dynamically typed:** everything with a name (variables, functions, etc) is given a type at runtime based on the thing's **current value**

What do we mean by runtime?

Translated: Converting source code into something a computer can understand (i.e. machine code)

Compiled vs **Interpreted**

Spot the difference: When is source code translated?

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

Compiled vs Interpreted: When is source code translated?

Dynamically typed, interpreted

- Types are 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)

Dynamic vs Static Typing

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
}
```

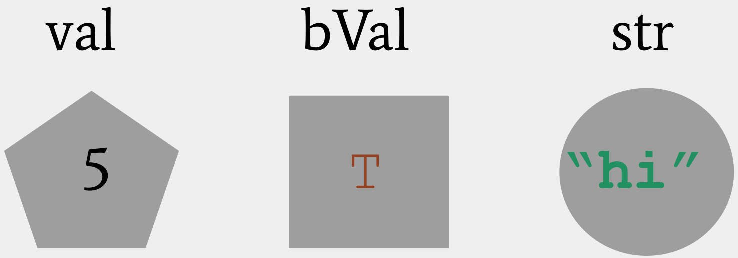
Dynamic vs Static Typing

Python

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

C++

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



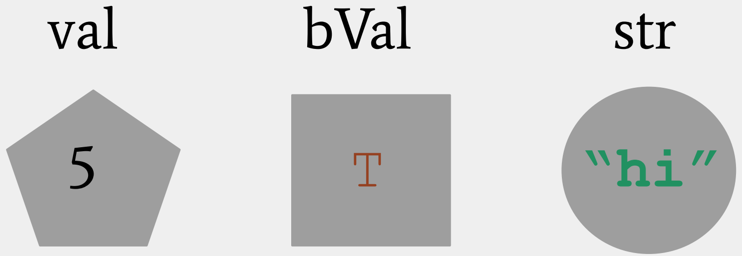
Dynamic vs Static Typing

Python

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

C++

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



Dynamic vs Static Typing

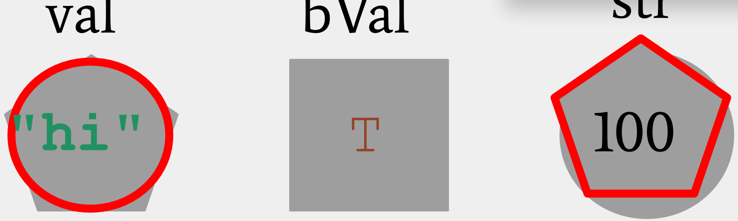
Python

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

C++

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

ERROR!



Dynamic vs Static Typing

Dynamic typing can cause some unexpected results!

Python

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

C++

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

Dynamic vs Static Typing

Dynamic typing can cause some unexpected results!

Python

```
def div_3(x):
    return x / 3
div_3("hello")
```

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

C++

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

//Compile error: this code will never run

Dynamic vs Static Typing

Python

```python
def mul_3(x):
    return x * 3
mul_3("10")
```

C++

```cpp
int mul_3(int x){
    return x * 3;
}
mul_3("10");
```

Dynamic vs Static Typing

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

```python
def add_3(x):
    return x + 3
add_3("10")
```

//returns “103”

C++

```cpp
int add_3(int x){
    return x + 3;
}
add_3("10");
```

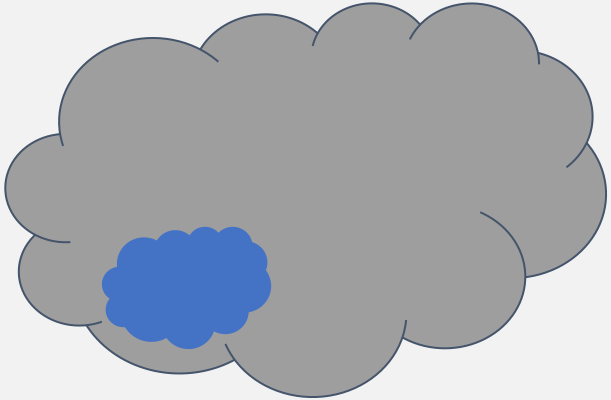
//Compile error: “10” is a string!
This code won’t run

Static typing makes your functions more readable!

Python

```
def div_3(x)
```

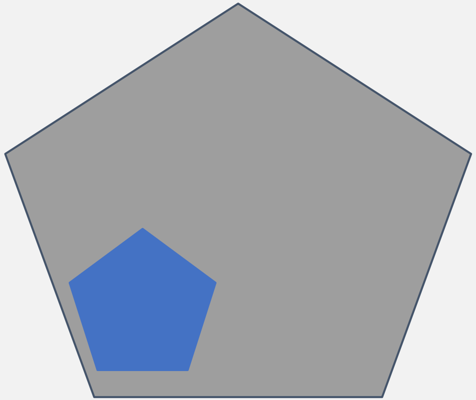
```
div_3: __ -> ??
```



C++

```
int div_3(int x)
```

```
div_3: int -> int
```



Types and Structs

http://web.stanford.edu/class/cs106l/

C++ to Python, probably

Types and Structs

http://web.stanford.edu/class/cs106l/

Fill in the blanks (again)!

```
int add(int a, int b);
    int, int -> int
string echo(string phrase);
______________________
string helloworld();
______________________
double divide(int a, int b);
______________________
```

Types and Structs

http://web.stanford.edu/class/cs106l/

Fill in the blanks (again)!

```
int add(int a, int b);
    int, int -> int
string echo(string phrase);
    string -> string
string helloworld();
_________________________
double divide(int a, int b);

_________________________
```

Types and Structs

http://web.stanford.edu/class/cs106l/

Fill in the blanks (again)!

```
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);

______________________
```

Fill in the blanks (again)!

```
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?

Aside: Function overloading

In C++, you cannot define multiple identical functions.

Aside: Function overloading

In C++, you cannot define multiple identical functions.

But what if we want two versions of a function for two different types?

- Example: int division vs double division

Aside: Function overloading

In C++, you cannot define multiple identical functions.

But what if we want two versions of a function for two different types?

- Example: int division vs double division

We can **overload** a function to have multiple versions!

Aside: Function overloading

```
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
```

To overload a function, declare multiple functions with the **same name** but **differently typed** parameters!

Aside: Function overloading

C++ allows specifying default parameter values!

```
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 ??
```

Aside: Function overloading

```
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?

Agenda

01. **C++ Basics**
Comparing to other languages you might know

02. **Types**
Working with a statically typed language

03. **Structs**
...and pairs and structured binding

The problem with types so far

Strongly and statically typed languages are great, but there are a few downsides:

- it can be a pain to know what the type of a variable is
- any given function can only have exactly one return type
- C++ primitives (and even the types in the STL) can be limited

The problem with types so far

Strongly and statically typed languages are great, but there are a few downsides:

- it can be a pain to know what the type of a variable is
- any given function can only have exactly one return type
- C++ primitives (and even the types in the STL) can be limited

Aside: the `auto` keyword

`auto`: a keyword used in lieu of type when declaring a variable that tells the compiler to **deduce the type.**

Aside: the `auto` keyword

`auto`: a keyword used in lieu of type when declaring a variable that tells the compiler to **deduce the type.**

- This is NOT the same as not having a type!
- The compiler is able to determine the type itself without being explicitly told.

Aside: the `auto` keyword

I hope it's an int...

<u>`auto`:</u> a keyword used in lieu of type when declaring a variable that tells the compiler to **deduce the type.**

- This is NOT the same as not having a type!
- The compiler is able to determine the type itself without being explicitly told.

Using auto

```
// What types are these?
auto a = 3;
auto b = 4.3;
auto c = 'X';
auto d = "Hello";
```

Using auto

```
// What types are these?
auto a = 3; // int
auto b = 4.3; // double
auto c = 'X'; // char
auto d = "Hello"; // char* (a C string)
```

Questions?

The problem with types so far

Strongly and statically typed languages are great, but there are a few downsides:

- it can be a pain to know what the type of a variable is
- any given function can only have exactly one return type
- C++ primitives (and even the types in the STL) can be limited

The problem with types so far

Strongly and statically typed languages are great, but there are a few downsides:

- it can be a pain to know what the type of a variable is
- any given function can only have exactly one return type
- C++ primitives (and even the types in the STL) can be limited

Let's address these with structs!

What is a struct?

A **struct** is a a group of **named variables**, each with their own type, that allows programmers to **bundle different types** together!

Structs in code

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

Student s;
s.name = "Haven";
s.state = "AR";
s.age = 21; // use . to access fields
```

Structs can pass around grouped information…

```
Student s;
s.name = "Haven";
s.state = "AR";
s.age = 21; // use . to access fields

void printStudentInfo(Student s) {
   cout << s.name << " from " << s.state;
   cout << " (" << s.age ")" << endl;
}
```

...or return grouped information!

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

Student foundStudent = randomStudentFrom("AR");
cout << foundStudent.name << endl; // Haven
```

…or return grouped information!

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

Student foundStudent = randomStudentFrom("AR");
cout << foundStudent.name << endl; // Haven
```

This syntax is a little clunky to initialize!

Let's abbreviate!

```
Student s;
s.name = "Haven";
s.state = "AR";
s.age = 21;

//is the same as ...
Student s = {"Haven", "AR", 21};
```

The STL has its own structs!

`std::pair:` An STL built-in struct with **two fields** of **any type**

The STL has its own structs!

<u>**`std::pair:`**</u> An STL built-in struct with **two fields** of **any type**

- **`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`**.

The STL has its own structs!

`std::pair:` An STL built-in struct with **two fields** of **any type**

- `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`.

```
std::pair<int, string> numSuffix = {1,"st"};

cout << numSuffix.first << numSuffix.second; //prints 1st
```

The STL has its own structs!

__`std::pair:`__ An STL built-in struct with **two fields** of **any type**

- **`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`**.

```
struct Pair {
    fill_in_type first;
    fill_in_type second;
};
```

Basically the struct looks like this!

Use std::pair to return success and result

```
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");
```

Use std::pair to return success and result

```
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");
```

We can use `std::make_pair` to avoid specifying the type!

Use std::pair to return success and result

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

auto output = lookupStudent("Julie");
```

auto makes this neater!

Questions?

Code Demo!

quadratic.cpp

In case you forgot...

You can write a general quadratic equation in the format of:

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

Which can then be solved for x as:

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

In case you forgot...

You can write a general quadratic equation in the format of:

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

Which can then be solved for x as:

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

Known as the <u>radical</u>!

Types and Structs

http://web.stanford.edu/class/cs106l/

In case you forgot...

You can write a general quadratic equation in the format of:

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

Which can then be solved for x as:

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

Known as the radical!

If the radical < 0, there are no real roots.

Recap:

- Everything with a name in your program has a **type**
- **Static type system** 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

Types and Structs

http://web.stanford.edu/class/cs106l/

Thanks!

Next up: Initialization and References!