四川大學
SICHUAN UNIVERSITY

Object Oriented Programming—C++
Lecture2 Types and Structs

Qijun Zhao
College of Computer Science
Sichuan University
Spring 2023

Agenda Today

- **Quick Recap**
- Types
- Intro to structs

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::**
- Extremely powerful and well-maintained

C++: Basic Syntax & the STL

```cpp
#include <iostream>

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

```cpp
#include "stdio.h"
#include "stdlib.h"

int main(int argc, char *argv) {
    printf("%s", "Hello, world!\n");
    // ^a C function!
    return EXIT_SUCCESS;
}
```

A note about STL naming conventions

- **STL** = Standard Template Library
  - Contains TONS of functionality (algorithms, containers, functions, iterators) some of which we will explore in this class
- The **namespace** for the STL is **std**
  - std is the abbreviation for standard
  - IDK why they didn't name the namespace stl
- So to **access elements** from the STL use **std::**

Today

- Quick Recap
- **Types**
- Intro to structs

C++ Fundamental Types

```
int val = 5;
char ch = 'F';
float decimalVal1 = 5.0;
double decimalVal2 = 5.0;
bool bVal = true;
```

C++ 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
```

C++ 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 = "Sarah";
```

Fill in the types!

```
______ 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;
}
```

Fill in the types!

```
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;
}
```

Fill in the types!

```
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;
}
```

```
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 vs. **Dynamically typed**

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

Dynamically typed: everything with a name (variables, functions, etc) is given a type **at runtime based on ?**

Runtime: Period when program is executing commands (after compilation, if compiled)

C++ is a statically typed language

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

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

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

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)

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 defined before program runs during compilation
- Example: C++

C++ Types in Action

```
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
}
```

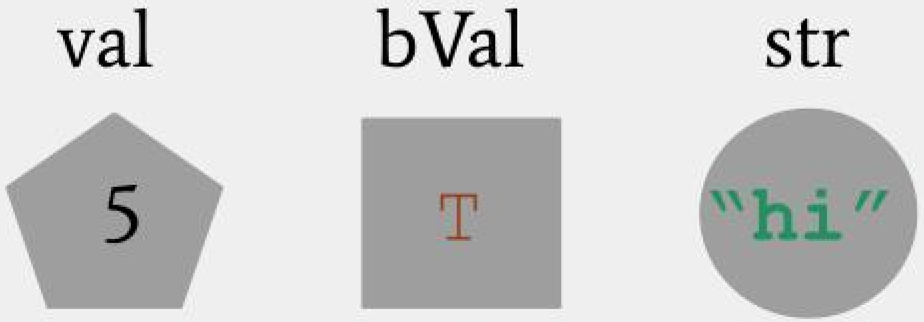
Dynamic vs Static typing: Python vs C++

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

```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;
```

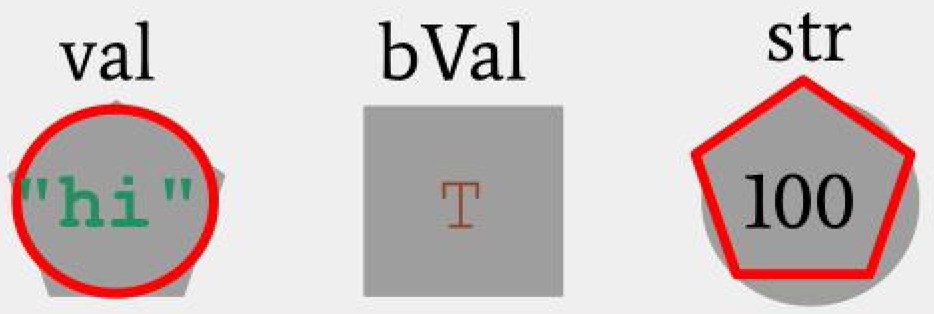
Python

```
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++

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

Dynamic vs Static typing: Python vs C++

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")
```

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")
//Compile error: this code
will never run
```

Dynamic vs Static typing: Python vs C++

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 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
```

Python

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

C++

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

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**

Python

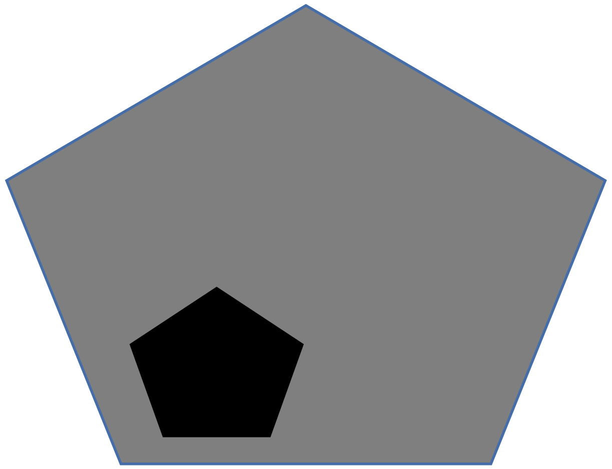
def div_3(x)

div_3: __ -> ??

C++

int div_3(int x)

div_3: int -> int



What are the "types" of the following functions?

```
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?

```
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?

```
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
```

Overloading

- What if we want two versions of a function for two different types?
- Example: int division vs double division

Define two functions with the same name but different types

```
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)
half(3.0)
```

Define two functions with the same name but different types

```
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 ?
```

Define two functions with the same name but different types

```
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
```

Define two functions with the same name but different types

```
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 ??
```

Define two functions with the same name but different types

```
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
```

Today

- Quick Recap
- Types
- **Intro to structs**

Struct: a group of named variables each with their own type. A way 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 = "Sarah";
s.state = "CA";
s.age = 21; // use . to access fields
```

Use structs to pass around grouped information

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

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

Use structs to return grouped information

```
Student randomStudentFrom(std::string state) {
  Student s;
  s.name = "Sarah";//random = always Sarah
  s.state = state;
  s.age = std::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

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

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

Definition

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
```

- **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;
};
```

Use std::pair to return success + 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");
```

To avoid specifying the types of a pair, use

std::make_pair(field1, field2)

```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");
```

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

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

Type Deduction 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)
auto e = std::make_pair(3, "Hello");
// 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.

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 and very cumbersome to write out

四川大學

SICHUAN UNIVERSITY

Coding for love,
Coding for the world

Qijun Zhao

qjzhao@scu.edu.cn