Lecture 2: Types and Structs

CS 106L, Fall ‘20
Today’s Agenda

- Recap: Types
- Structs
- Type Deduction with Auto
- Structured Binding
Namespaces

- Some 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!
Recap: Types
C++ is a **static-typed** language

**Python**

```python
a = 3
b = "test"

def func(c):
    # do something
```

**C++**

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

void func(c) {
    // do something
```
int a = 3;
string b = "test";

void func(double c) { // vars take type
    std::cout << b << std::endl; // doesn’t return anything, so it’s void
}

if (b == 3) { // don’t have to repeat types during use
    // do something
}
Why static typing?

- Better performance
- Easier to understand
- Better error checking
Catches errors at compile time

```python
def add_3(x):
    return first + 3

add_3("10")  # whoops, that's a string! returns "103"...
```

```c
int add_3(int x) {
    return first + 3;
}
```

```python
add_3("10");  # caught as compiler error!
```
Common Types

int  5
double  77.3
string  "test"
bool  true
size_t  5  // non-negative
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?

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;
}
**Overloading**

Define two functions with the same name but different call signature

```c
double func(int x) {    // (1)
    return (double) x + 3;  // typecast: int → double
}

double func(double x) {    // (2)
    return x * 3;
}

func(2)       // uses version (1), returns ?
func(2.0)     // uses version (2), returns ?
```

Answer in the chat!
Overloading

Define two functions with the same name but different call signature

```c
double func(int x) {   // (1)
    return (double) x + 3;  // typecast: int → double
}

double func(double x) {   // (2)
    return x * 3;
}

func(2)     // uses version (1), returns 5
func(2.0)    // uses version (2), returns 15.0
```
Every student has:
  ○ A name (string)
  ○ A home state (string)
  ○ An age (int)
A bigger problem

```java
___ getStudentWithID(int id) {
    // how can we return a string, a string, and an int?
}
```

// python:
// return ("a", "b", 3)

How can we return multiple things?
void printStudentInfo(string name, string state, int age) {
    cout << name << " from " << state;
    cout << " (" << age << ")" << endl;
}

void enrollStudent(string name, int state, int age) {
    // do something...
}
Structs
A **struct** is a group of named variables

*each with their own type*
struct Student {
    string name;       // these are called fields
    string state;     // separate these by semicolons
    int age;
}

Student s;
s.name = "Ethan";    // use the . operator to access fields
s.state = "CA";
s.age = 20;
Structs let you group information together.
Pass around information together

```cpp
Student s;
s.name = "Ethan"; // use the . operator to access fields
s.state = "CA";
s.age = 20;

void printStudentInfo(Student student) {
    cout << student.name << " from " << student.state;
    cout << " (" << student.age "\)" << endl;
}
```
Return information together

```cpp
Student lookupStudent() {
    Student s;
    s.name = "Ethan";
    s.state = "CA";
    s.age = 20;
    return s;
}

Student foundStudent = lookupStudent();
cout << foundStudent.name << endl;  // Ethan
```
Abbreviated Struct Notation

Student s;
s.name = “Ethan”;
s.state = “CA”;
s.age = 20;

Student s = {“Ethan”, “CA”, 30};
// note that order is based on the original struct order!
// generally prefer this syntax for initialization
Questions?
Pairs
(and maybe tuples)
A **pair** is a struct with two fields.

```cpp
int main() {
    std::pair<bool, Student> query_result;
    query_result.first = true;
    Report current = query_result.second;
}
```

📝 **std::pair** is a **template**. You can use any type inside it; type goes in the `< >`. (We'll learn more about templates in a future lecture.)
Common use case: return success + result

```cpp
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("Keith");
```

*std::make_pair* is a generic way to make a pair without explicitly writing a type!
A **tuple** is a struct with lots of fields.

```cpp
int main() {
    std::tuple<string, int, int> query_result;
    string name = std::get<1>(query_result);
    int num = std::get<2>(query_result);
}
```

- **std::tuple** is uncommon. We won’t use it much in future lectures. (Datatypes like **std::vector** are generally more useful.)
- Yes, **std::get** is a template. Don’t worry too much about this.
Live 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:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$
Let's return a std::pair of the form <does_solution_exist, <sol1, sol2>>:

```cpp
std::pair<bool, std::pair<double, double>> quadratic (int a, int b, int c) {
    double inside = b*b - 4*a*c;
    std::pair<double, double> blank;
    if (inside < 0) return std::make_pair(false, blank);

    std::pair<double, double> answer = std::make_pair(( -b+sqrt(inside))/2,
                                                        ( -b-sqrt(inside))/2 );
    return std::make_pair(true, answer);
}
```
Here's how we would use this:

```cpp
int main() {
    int a, b, c;
    std::cin >> a >> b >> c;  // this gets input
    std::pair<bool, std::pair<double, double>> result = quadratic(a, b, c);
    if (result.first) {
        std::pair<double, double> solutions = result.second;
        std::cout << solutions.first << solutions.second << std::endl;
    } else {
        std::cout << "No solutions found!" << std::endl;
    }
}
```
Type Deduction with auto
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*>
Type Deduction using `auto`

🤔 Don’t be ambiguous or the compiler won’t get what you mean...

```c++
auto wrong(); // this won’t work
void wrong(string a, auto b) { // neither will this work
    return a * b;
}
```
When should we use auto?
int main() {
    int a, b, c;
    std::cin >> a >> b >> c;
    std::pair<bool, std::pair<double, double>> result = quadratic(a, b, c);
    bool found = result.first;
    if (found) {
        std::pair<double, double> solutions = result.second;
        std::cout << solutions.first << solutions.second << std::endl;
    } else {
        std::cout << "No solutions found!" << std::endl;
    }
}
int main() {
    int a, b, c;
    std::cin >> a >> b >> c;
    auto result = quadratic(a, b, c);
    bool found = result.first;
    if (result.first) {
        auto solutions = result.second;
        std::cout << solutions.first << solutions.second << std::endl;
    } else {
        std::cout << "No solutions found!" << std::endl;
    }
}
Don’t overuse `auto`

...but use it to shorten long types
Structured Binding
Structured binding lets you initialize directly from the contents of a struct.

Before

```cpp
auto p = std::make_pair("s", 5);
string a = s.first;
int b = s.second;
```

After

```cpp
auto p = std::make_pair("s", 5);
auto [a, b] = p;
// a is of type string
// b is of type int
```

This works for regular structs, too. Also, no nested structured binding.
A better way to use quadratic

Let’s apply structured binding:

```cpp
int main() {
    int a, b, c;
    std::cin >> a >> b >> c;
    auto [found, solutions] = quadratic(a, b, c);
    if (found) {
        auto [x1, x2] = solutions;
        std::cout << x1 << " " << x2 << endl;
    } else {
        std::cout << "No solutions found!" << endl;
    }
}
```

📝 This is better is because it’s semantically clearer: variables have clear names.