Structures

CS 106L Spring 2020 – Avery Wang and Anna Zeng
Stanford University

6 April 2020

Game Plan

- aggregate structures
- type deduction
- basics of C++ libraries

C++ details we will cover

Language

- structs
- auto + structured binding

Library

- std::pair, std::tuple
- std::vector
- many examples of each

Key questions we will answer today

- what are some basic data types in the STL?
- what are some techniques to handle complicated types?
- how do we construct, initialize, and use these types?

Aggregate Structures

A struct is a collection of named variables.

```
1.  struct Report {
2.    string date;
3.    size_t cases;
4.    size_t deaths;
5.  }; // don't forget the semicolon!
6.
7.  int main() {
8.    Report current;
9.    current.date = "2020-03-30";
10.   current.cases = 715551;
11.   current.deaths = 33656;
12. }
```

Structs are similar to classes, except all members (2-4) are public.

A size_t (3-4) is a non-negative int.

You can access each member (9-11) using the "." operator.

sidenote: structs are part of the C language

pair and tuple are structs with standardized names.

```
1.  int main() {
2.    std::pair<bool, Report> query_result;
3.    query_result.first = true;
4.    Report current = query_result.second;
5.
6.    std::tuple<string, size_t, size_t> report;
7.    size_t cases = std::get<1>(report);
8.  }
```

An std::pair is a struct with members first and second. (3-4)

pair and tuple are templates. You put the types of each member in the brackets (2, 6).

pair appears a lot in the STL. tuple is rarely used so we won't talk much about it.

sidenote: std::get<T> is a template - don't worry too much about this.

std::array and std::vector are collections of homogeneous type.

```
1.  int main() {
2.    std::array<int, 2> arr; // {0, 0}
3.    arr[0] = 10;            // {10, 0}
4.    arr[1] = 2;             // {10, 2}
5.    cout << vec[-1];        // undefined behavior
6.
7.    std::vector<int> vec;   // {}
8.    vec.push_back(1);       // {1}
9.    vec.resize(3);          // {1, 0, 0}
10.   vec[2] = 1;             // {1, 0, 2}
11.   cout << vec[3];         // undefined behavior
12. }
```

A std::array (2) has a fixed size decided at compile-time, templatized with type and size.

A std::vector (7) has a dynamic size, which changes as elements are added.

C++ does not perform bounds checking when indexing! (5, 11)

We will go into much more detail with std::vector when we discuss STL collections. It's just here to give you an idea of what structures are - collections of elements.

Summary of Structures

std::pair<T1, T2>

std::tuple<Args…>

std::array<T, n>

std::vector<T>

Summary of Structures

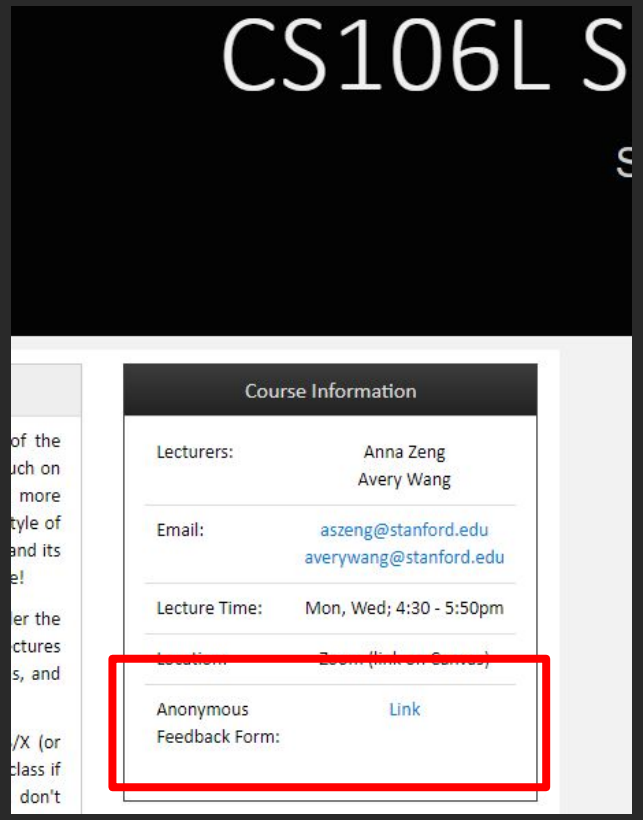
pair, tuple, array have fixed number of elements.

vector has variable number of elements.

Logistics

Logistics

- Join the Piazza if you haven't already!
  - https://piazza.com/stanford/spring2020/cs106l
- Fill out the intro survey!
  - https://forms.gle/MahBUdB54mfqWnQQ6
  - Due Monday, April 13, 11:59 pm
- New: Anonymous feedback form on website!



Logistics

- Office hours this quarter:
  - Assignment office hours TBA
  - Private post on Piazza to arrange other times
  - Anna will be holding "command-line office hours" this week!
    - Thurs (4/9) 9:30-10:15 am PST
    - Thurs (4/9) 8-9 pm PST
    - Feel free to come with any other questions as well

2-min stretch break!

auto and structured binding

the compiler will deduce the type for you using auto.

```
1.  auto a = 3;
2.  auto b = 4.3;
3.  auto c = 'X';
4.  auto d = "Hello";
5.  auto e = "Hello"s;
6.  auto f = std::make_pair(3, "Hello");
7.  auto g = {1, 2, 3};
8.  auto h = [](int i) {return 3*i;};
```

Sidenote: auto discards cv-qualifiers and references. We'll get to that at the end of this lecture, maybe next time.

Answers:

a = int, b = double, c = char,
d = char*, e = std::string,
f = std::pair<int, char*>,
g = std::initializer_list<int>
h = [known only by compiler]

Key takeaways:

- compiler is smart but can't read your mind.

- don't be ambiguous!

- sometimes auto must be used (in the case of lambdas)

When and why to use auto?

When?

- You don't care about the exact type (iterators).
- When its type is clear from context (templates).
- When you can't figure out the type (lambdas).
- Avoid using auto for return values (exception: generic programming)

Why?

- Correctness: no implicit conversions, uninitialized variables.
- Flexibility: code easily modifiable if type changes need to be made.
- Powerful: very important when we get to templates!
- Modern IDE’s (eg. Qt Creator) can infer a type simply by hovering your cursor over any auto, so readability not an issue!

Sidenote: auto discards cv-qualifiers and references. We'll get to that at the end of this lecture, maybe next time.

unpack aggregate structures using structured binding!

```
1.  auto p = std::make_pair(true, 3);
2.  auto [found, num] = p;
3.
4.  auto arr = std::make_tuple('x','y','z','w');
5.  auto [a, b, c, d] = arr;
```

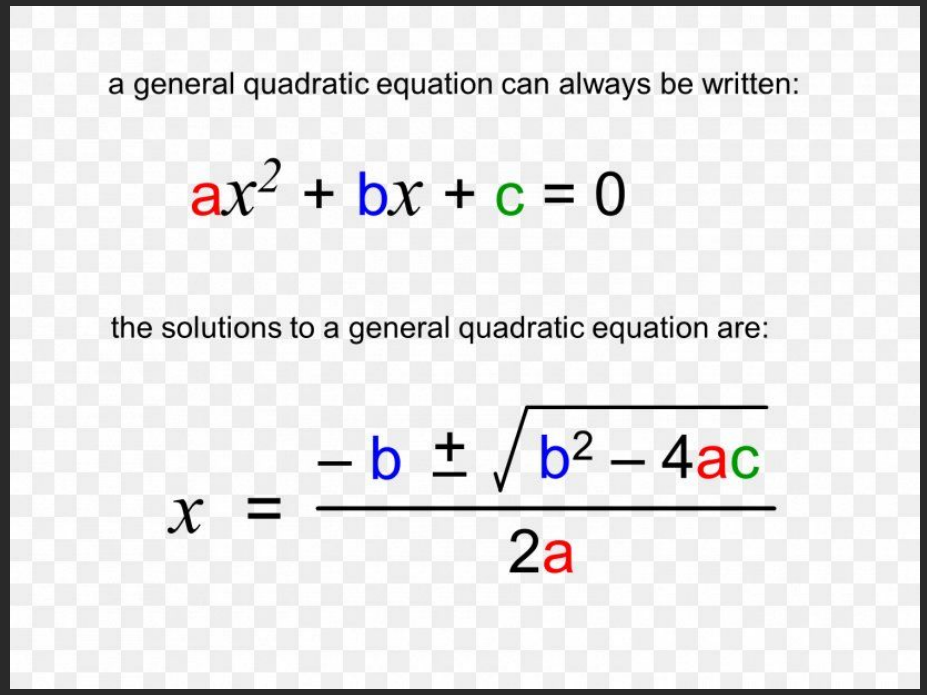
Creators (a.k.a. make_X) allows you to create aggregate structures without ever figuring out the types.

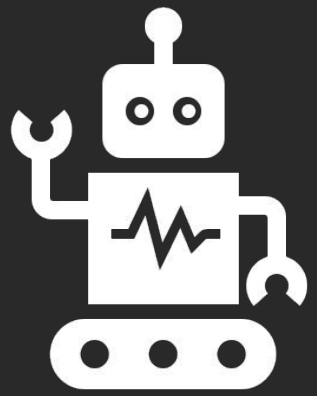
Structured binding (C++17) lets you unpack each member into a variable.

Same comment for cv-qualifiers and references still apply!

in class exercise: quadratic solver

```
1.  int main() {
2.    int a, b, c;
3.    std::cin >> a >> b >> c;
4.
5.    /* you decide */ = quadratic(a, b, c);
6.    // print the solutions we found
7.    // or state no solutions
8.  }
9.
10. [something] quadratic(int a, int b, int c) {
11.   // implement this
12. }
```





Example

quadratic equation solver returning multiple things

let's return multiple things in a single function

```
1.  std::pair<bool, std::pair<double, double>> quadratic(int a, int b, int c) {
2.    double D = b*b - 4*a*c;
3.    std::pair<double, double> blank;
4.    if (D < 0) return std::make_pair(false, blank);
5.
6.    std::pair<double, double> answer =
7.              std::make_pair( (-b+sqrt(D))/2, -b-sqrt(D))/2 );
8.    return std::make_pair(true, answer);
9.  }
```

std::make_pair is a generic way to make a pair without having to explicitly type out a type!

here's how we would use the quadratic function

```
1.  int main() {
2.    int a, b, c;
3.    std::cin >> a >> b >> c;
4.
5.    auto [found, solutions] = quadratic(a, b, c);
6.    if (found) {
7.      auto [x1, x2] = solutions;
8.      std::cout << x1 << " " << x2;
9.    } else {
10.     std::cout << "No solutions";
11.   }
12. }
```

Note: unfortunately nested structured binding isn't possible.

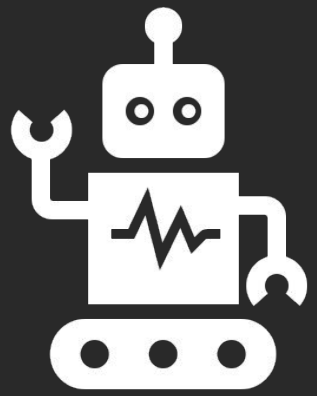
Common pattern you'll see: the first boolean represents if something was found (5, 7).

Line 3 is simply to read input, feel free to ignore.

Key considerations

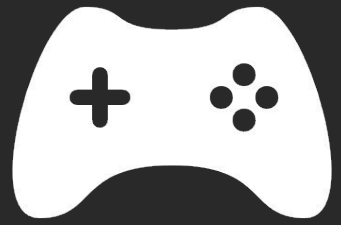
Assuming CS 106B has already covered references:

- why is returning a std::pair superior to using reference parameters or returning a std::vector of size 2?
- are there efficiency concerns with return vs. references?



Homework

Set Up Qt Creator



Next time

References