Initialization & References

Fun times!

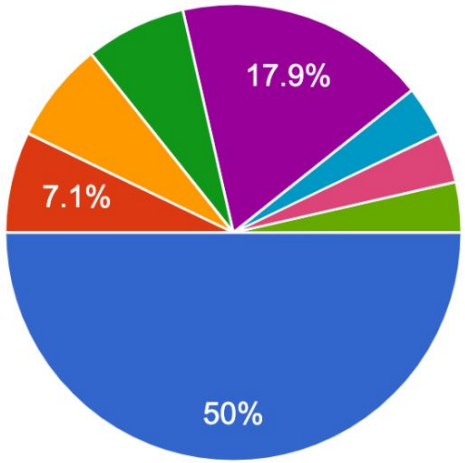
Attendance

bit.ly/3mocA0n

Intro Survey

Year

28 responses

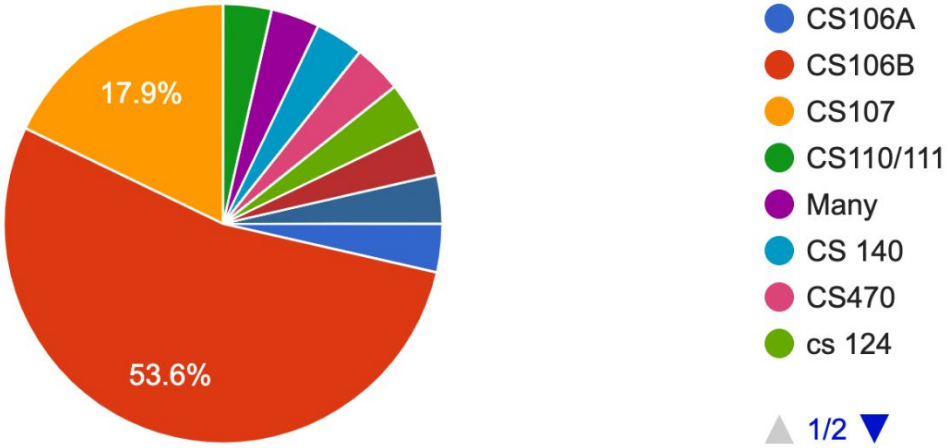


- Freshman
- Sophomore
- Junior
- Senior
- Coterm/MS
- Masters - CCRMA (Music)
- I came in as class of 2025, but I tore my ACL last Spring, which had me take a year leave of absence. This is my first…
- PhD

Intro Survey

What's the highest level coding focused CS class you've taken (or are currently enrolled in)?

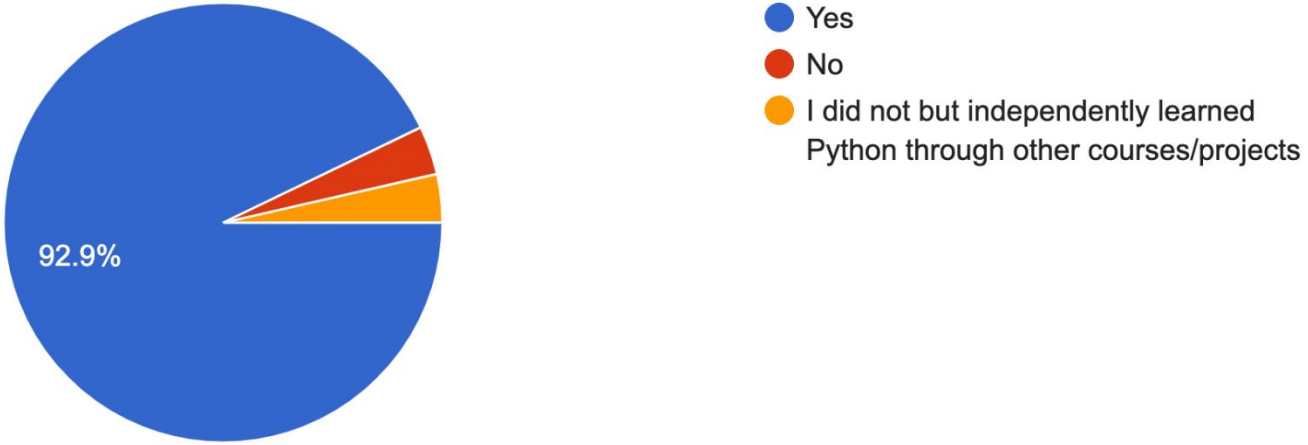
28 responses



Intro Survey

Did you take CS106A (or have similar experience with Python)?

28 responses



Intro Survey

pre-covid I went to costco 3 times a week as an evening walk

I like cooking and one reason of applying coterm is getting rid of the meal plan :D

I am the Bookstore's #1 candy purchaser.

I and 2/3 of my siblings are born on the 18th of separate months.

Intro Survey

I love crocheting :D I am currently working on a large Miffy plushie.

i took a class with haven last quarter (boring)

I've also gone skydiving!

- Blew air through my nose when answering the previous question

Announcements

Announcements

- Office hours times posted on class website!
  - Sarah: Tuesday 3:15 - 4:15pm in Thornton 207
  - Haven: Thursday 3:15 - 4:15pm in Thornton 208

A note about feedback

- We welcome feedback! This class is meant for you.
  - Always welcome to send us an email, make an Ed post, or talk to us after class or in office hours
  - If you want to provide feedback anonymously, we created an anonymous feedback form (also posted on Ed)

Today

- **Initialization**
- Using auto
- References
- If time: Const

Definition

Initialization: How we provide initial values to variables

Reminder: 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
```

Recall: Two ways to initialize a struct

```
Student s; // initialization after we declare
s.name = "Sarah";
s.state = "CA";
s.age = 21;
//is the same as ...
Student s = {"Sarah", "CA", 21};
// initialization while we declare
```

Multiple ways to initialize a pair...

```
std::pair<int, string> numSuffix1 = {1,"st"};

std::pair<int, string> numSuffix2;

numSuffix2.first = 2;

numSuffix2.second = "nd";

std::pair<int, string> numSuffix2 =

                        std::make_pair(3, "rd");
```

Definition

Uniform initialization: curly bracket initialization. Available for all types, immediate initialization on declaration!

Uniform Initialization

```
std::vector<int> vec{1,3,5};

std::pair<int, string> numSuffix1{1,"st"};

Student s{"Sarah", "CA", 21};

// less common/nice for primitive types, but possible!

int x{5};

string f{"Sarah"};
```

Careful with Vector initialization!

```
std::vector<int> vec1(3,5);

// makes {5, 5, 5}, not {3, 5}!

// uses a std::initializer_list (more later)

std::vector<int> vec2{3,5};

// makes {3, 5}
```

CODE DEMO

TLDR: use uniform initialization to initialize every field of your non-primitive typed variables - but be careful not to use vec(n, k)!

Questions?

Today

- ~~Initialization~~
- **Using auto**
- References
- If time: Const

Recap: Type Deduction with `auto`

Definition

`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");
```

📝 `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; // 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.

‼️ `auto` **does not mean that the variable doesn’t have a type.**

It means that the type is **deduced** by the compiler.

When should we use auto?

Code Demo Recap!

quadratic.cpp

a general quadratic equation can always be written:

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

Radical

the solutions to a general quadratic equation are:

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

If Radical < 0, no real roots

Quadratic: Typing these types out is a pain...

```
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 << endl;
    } else {
        std::cout << "No solutions found!" << endl;
    }
}
```

Quadratic: Typing these types out is a pain...

```
int main() {
    int a, b, c;
    std::cin >> a >> b >> c;
    auto result = quadratic(a, b, c);
    bool found = result.first;
    if (found) {
        auto solutions = result.second;
        std::cout << solutions.first << solutions.second << endl;
    } else {
        std::cout << "No solutions found!" << endl;
    }
}
```

Cleaner! 🧼

😊

Don’t overuse `auto`

Don’t overuse auto!

```
int main() {
    auto a, b, c;
    std::cin >> a >> b >> c;
    auto result = quadratic(a, b, c);
    bool found = result.first;
    if (found) {
        auto solutions = result.second;
        std::cout << solutions.first << solutions.second << endl;
    } else {
        std::cout << “No solutions found!” << endl;
    }
}
```

Can't deduce the type b/c no value provided

```
int main() {
    auto a, b, c; //compile error!
    std::cin >> a >> b >> c;
    auto result = quadratic(a, b, c);
    bool found = result.first;
    if (found) {
        auto solutions = result.second;
        std::cout << solutions.first << solutions.second << endl;
    } else {
        std::cout << "No solutions found!" << endl;
    }
}
```

ERROR!

For simple types (like bool) type it out

```
int main() {
    int a, b, c;
    std::cin >> a >> b >> c;
    auto result = quadratic(a, b, c);
    auto found = result.first; //code less clear :/
    if (found) {
        auto solutions = result.second;
        std::cout << solutions.first << solutions.second << endl;
    } else {
        std::cout << "No solutions found!" << endl;
    }
}
```

LESS CLEAR 👎☹️

Don’t overuse `auto`

...but use it to reduce long type names

Questions?

Structured Binding

Structured binding lets you initialize directly from the contents of a struct

Before

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

After

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

📝 This works for regular structs, too. Also, no nested structured binding.

A better way to use quadratic...

```
int main() {
    int a, b, c;
    std::cin >> a >> b >> c;
    auto result = quadratic(a, b, c);
    bool found = result.first;
    if (found) {
        auto solutions = result.second;
        std::cout << solutions.first << solutions.second << endl;
    } else {
        std::cout << "No solutions found!" << endl;
    }
}
```

Using Structured Binding

```
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.

Questions?

Today

- ~~Initialization~~
- ~~Using auto~~
- **References**
- If time: Const

Definition

Reference: An alias (another name) for a named variable

References in 106B

```
void changeX(int& x){ // changes to x will persist
    x = 0;
}
void keepX(int x){
    x = 0;
}

int a = 100;
int b = 100;

changeX(a); // a becomes a reference to x
keepX(b);   // b becomes a copy of x

cout << a << endl; //0
cout << b << endl; //100
```

Standard C++ vector (intro)

Stanford Vector vs Standard std::vector

```
Vector<int> v;
Vector<int> v(n, k);
v.add(k);
v[i] = k;
int k = v[i];

v.isEmpty();
v.size();
v.clear();
v.insert(i, k);
v.remove(i);
```

```
std::vector<int> v;
std::vector<int> v(n, k);
v.push_back(k);
v[i] = k;
int k = v[i];

v.empty();
v.size();
v.clear();
// stay tuned
// stay tuned
```

References to variables

```
vector<int> original{1, 2};
vector<int> copy = original;
vector<int>& ref = original;
original.push_back(3);
copy.push_back(4);
ref.push_back(5);

cout << original << endl;
cout << copy << endl;
cout << ref << endl;
```

References to variables

```
vector<int> original{1, 2};
vector<int> copy = original;
vector<int>& ref = original;
original.push_back(3);
copy.push_back(4);
ref.push_back(5);

cout << original << endl; // {1, 2, 3, 5}
cout << copy << endl;
cout << ref << endl;
```

References to variables

```
vector<int> original{1, 2};
vector<int> copy = original;
vector<int>& ref = original;
original.push_back(3);
copy.push_back(4);
ref.push_back(5);

cout << original << endl; // {1, 2, 3, 5}
cout << copy << endl;     // {1, 2, 4}
cout << ref << endl;
```

References to variables

```
vector<int> original{1, 2};
vector<int> copy = original;
vector<int>& ref = original;
original.push_back(3);
copy.push_back(4);
ref.push_back(5);

cout << original << endl; // {1, 2, 3, 5}
cout << copy << endl;     // {1, 2, 4}
cout << ref << endl;      // {1, 2, 3, 5}
```

References to variables

```
vector<int> original{1, 2};
vector<int> copy = original;
vector<int>& ref = original;
original.push_back(3);
copy.push_back(4);
ref.push_back(5);

cout << original << endl; // {1, 2, 3, 5}
cout << copy << endl;     // {1, 2, 4}
cout << ref << endl;      // {1, 2, 3, 5}
```

"=" automatically makes a copy! Must use & to avoid this.

The classic reference-copy bug 1.0:

```
void shift(vector<std::pair<int, int>>& nums) {
   for (size_t i = 0; i < nums.size(); ++i) {
      auto [num1, num2] = nums[i];
      num1++;
      num2++;
   }
}
```

The classic reference-copy bug 1.0:

```
void shift(vector<std::pair<int, int>>& nums) {
    for (size_t i = 0; i < nums.size(); ++i) {
        auto [num1, num2] = nums[i];
    }
}
```

size_t is commonly used for indices because it's **unsigned and dynamically sized (using sizeof())**. Nitty gritty

++i: increment then return
i++: return then increment
In for loops, **both work the same and no performance** difference anymore so use what you prefer! Nitty gritty

The classic reference-copy bug 1.0:

```
void shift(vector<std::pair<int, int>>& nums) {
   for (size_t i = 0; i < nums.size(); ++i) {
      auto [num1, num2] = nums[i];
      num1++;
      num2++;
   }
}
```

2 min: THINK, PAIR, SHARE!

The classic reference-copy bug 1.0:

```
void shift(vector<std::pair<int, int>>& nums) {
   for (size_t i = 0; i < nums.size(); ++i) {
      auto [num1, num2] = nums[i];
      num1++;
      num2++;
   }
}
```

This creates a copy of the course

This is updating that same copy!

The classic reference-copy bug 1.0: Fixed

```
void shift(vector<std::pair<int, int>>& nums) {
   for (size_t i = 0; i < nums.size(); ++i) {
      auto& [num1, num2] = nums[i];
      num1++;
      num2++;
   }
}
```

The classic reference-copy bug 2.0:

```
void shift(vector<std::pair<int, int>>& nums) {
   for (auto [num1, num2]: nums) {
      num1++;
      num2++;
   }
}
```

The classic reference-copy bug 2.0:

```
void shift(vector<std::pair<int, int>>& nums) {
    for (auto [num1, num2]: nums) {
        num1++;
        num2++;
    }
}
```

This is updating that same copy!

This creates a copy of the course

The classic reference-copy bug 2.0, fixed:

```
void shift(vector<std::pair<int, int>>& nums) {
    for (auto& [num1, num2]: nums) {
        num1++;
        num2++;
    }
}
```

Definition: l-values vs r-values

- **l-values** can appear on the **left** or **right** of an =
- `x` is an **l-value**

```
int x = 3;
int y = x;
```

l-values have names

l-values are **not temporary**

Definition: l-values vs r-values

- **l-values** can appear on the **left** or **right** of an =
- `x` is an **l-value**

```
int x = 3;
int y = x;
```

l-values have names

l-values are **not temporary**

- **r-values** can ONLY appear on the **right** of an =
- `3` is an **r-value**

```
int x = 3;
int y = x;
```

r-values don't have names

r-values are **temporary**

The classic reference-rvalue error

```
void shift(vector<std::pair<int, int>>& nums) {
    for (auto& [num1, num2]: nums) {
        num1++;
        num2++;
    }
}

shift({{1, 1}});
```

The classic reference-rvalue error

```
void shift(vector<std::pair<int, int>>& nums) {
    for (auto& [num1, num2]: nums) {
        num1++;
        num2++;
    }
}

shift({{1, 1}});
// {{1, 1}} is an rvalue, it can't be referenced
```

The classic reference-rvalue error, fixed

```
void shift(vector<pair<int, int>>& nums) {
   for (auto& [num1, num2]: nums) {
      num1++;
      num2++;
   }
}
auto my_nums = {{1, 1}};
shift(my_nums);
```

Note: You can only create references to variables

```
int& thisWontWork = 5; // This doesn't work!
```

Questions?

Today

- ~~Initialization~~
- ~~Using auto~~
- ~~References~~
- **If time: Const**

BONUS: Const and Const References

`const` indicates a variable can't be modified!

`const` variables can be references or not!

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};  // a const variable
std::vector<int>& ref = vec;         // a regular reference
const std::vector<int>& c_ref = vec;  // a const reference

vec.push_back(3);
c_vec.push_back(3);
ref.push_back(3);
c_ref.push_back(3);
```

`const` indicates a variable can't be modified!

`const` variables can be references or not!

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};  // a const variable
std::vector<int>& ref = vec;         // a regular reference
const std::vector<int>& c_ref = vec;  // a const reference

vec.push_back(3);    // OKAY
c_vec.push_back(3);
ref.push_back(3);
c_ref.push_back(3);
```

`const` indicates a variable can't be modified!

`const` variables can be references or not!

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};  // a const variable
std::vector<int>& ref = vec;         // a regular reference
const std::vector<int>& c_ref = vec;  // a const reference

vec.push_back(3);    // OKAY
c_vec.push_back(3);  // BAD - const
ref.push_back(3);
c_ref.push_back(3);
```

`const` indicates a variable can't be modified!

`const` variables can be references or not!

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};  // a const variable
std::vector<int>& ref = vec;         // a regular reference
const std::vector<int>& c_ref = vec;  // a const reference

vec.push_back(3);    // OKAY
c_vec.push_back(3);  // BAD - const
ref.push_back(3);   // OKAY
c_ref.push_back(3);
```

`const` indicates a variable can't be modified!

`const` variables can be references or not!

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};  // a const variable
std::vector<int>& ref = vec;         // a regular reference
const std::vector<int>& c_ref = vec;  // a const reference

vec.push_back(3);    // OKAY
c_vec.push_back(3);  // BAD - const
ref.push_back(3);   // OKAY
c_ref.push_back(3); // BAD - const
```

Can’t declare non-const reference to const variable!

```
const std::vector<int> c_vec{7, 8};  // a const variable

// BAD - can't declare non-const ref to const vector
std::vector<int>& bad_ref = c_vec;
```

Can’t declare non-const reference to const variable!

```
const std::vector<int> c_vec{7, 8};  // a const variable

// fixed
const std::vector<int>& bad_ref = c_vec;
```

Can’t declare non-const reference to const variable!

```
const std::vector<int> c_vec{7, 8};  // a const variable

// fixed
const std::vector<int>& bad_ref = c_vec;

// BAD - Can't declare a non-const reference as equal
// to a const reference!
std::vector<int>& ref = c_ref;
```

const & subtleties

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};

std::vector<int>& ref = vec;
const std::vector<int>& c_ref = vec;

auto copy = c_ref;          // a non-const copy
const auto copy = c_ref;    // a const copy
auto& a_ref = ref;          // a non-const reference
const auto& c_aref = ref;   // a const reference
```

Remember: C++, by default, makes copies when we do variable assignment! We need to use & if we need references instead.

When do we use references/const references?

- If we're working with a variable that takes up little space in memory (e.g. int, double), we don't need to use a reference and can just copy the variable
- If we need to alias the variable to modify it, we can use references
- If we don't need to modify the variable, but it's a big variable (e.g. std::vector), we can use const references

You can return references as well!

```
// Note that the parameter must be a non-const reference to return
// a non-const reference to one of its elements!
int& front(std::vector<int>& vec) {
    // assuming vec.size() > 0
    return vec[0];
}

int main() {
    std::vector<int> numbers{1, 2, 3};
    front(numbers) = 4; // vec = {4, 2, 3}
    return 0;
}
```

CODE DEMO

Can also return const references

```
const int& front(std::vector<int>& vec) {
    // assuming vec.size() > 0
    return vec[0];
}
```

Questions?

Recap:

- **Uniform Initialization**
  - A “uniform” way to initialize variables of different types!
- **References**
  - Allow us to alias variables
- **Const**
  - Allow us to specify that a variable can’t be modified

Thanks for coming!

Next time: Streams!