Lecture 11
Intro to C++

CS211 – Fundamentals of Computer Programming II

Branden Ghena – Fall 2021

Slides adapted from:
Jesse Tov

Reminder: relative homework difficulties

Homework	Difficulty
Hw01	2
Hw02	5
Hw03	7
Hw04	11
Hw05	6
Hw06	9
Final Project	10ish*

Hw04 is the last in C
one week break
Hw05 is the first in C++

* But really it's up to you

Administrivia

- Nothing is due until Sunday of this week
  - Lab05, which sets up your C++ environment
  - I'll try to publish this tonight

- Great time to catch up on any concepts you're still muddy about
  - Office hours are mostly empty, but course staff is still there!

Today's Goals

- Introduce C++
  - Goals of the language
  - Basics of how to use it
- Explore some key differences from C
  - Standard I/O
  - References
- Discuss a C++ data structure library: vector

Getting the code for today

- Download code in a zip file from here: https://nu-cs211.github.io/cs211-files/lec/09_introCPP.zip

- Extract code wherever

- Open with CLion
  - Make sure you open the folder with the CMakeLists.txt

  - Details on CLion in Lab05

Outline

- **Why C++?**
- Simple C++ I/O
- Pass-by-reference
- Vectors

What is C++?

- Feared by many; loved by few; understood by one
  - Bjarne Stroustrup, its designer

- Originally an extension to C called "C with Classes"
- Intended to bring modern (1980s) abstraction mechanisms to C
  - Data hiding
  - Generics

- Adds many other things too:
  - Destructors, Exceptions, Lambda, Dynamic Dispatch, Inheritance, Libraries
- But without slowing things down
  - "Pay (for language features) as you go"

What is C++ used for?

- Many different software areas
  - Browsers: Firefox, Chrome, Edge
  - Interactive software tools: Microsoft Office, Adobe Suite, AutoCAD
  - Language runtimes: Node.js, .NET, Java VMs
  - Major web services: Spotify, YouTube, Bloomberg’s financial database
  - Databases: Oracle, MySQL, IBM DB2, MongoDB, SQL Server
  - Game engines: Creation (Skyrim, Fallout), Frostbite (Battlefield, FIFA), Unreal

What is C++ used for?

- Many different software areas
  - Browsers: Firefox, Chrome, Edge
  - Office tools: Microsoft Office, Adobe Suite, AutoCAD
  - Language runtimes: Node.js, .NET, Java VMs
  - Major web services: Spotify, YouTube, Bloomberg's financial database
  - Databases: Oracle, MySQL, IBM DB2, MongoDB, SQL Server
  - Game engines: Creation (Skyrim, Fallout), Frostbite (Battlefield, FIFA), Unreal
- Generally:
  - Writing, big complicated programs that need to perform well
- You could write them in C, but C++ is more flexible, less work, and provides better ways to manage complexity

Why is CS211 using C++?

- The second half of CS211 focuses on learning to build larger programs and structure them using abstraction mechanisms

- Other popular languages that have the features we want don't let you take advantage of your newly-acquired C skills
  - Java, C#, Kotlin
  - And we do want to teach a *popular* language

- C++ lets you build larger programs with abstractions
  - But the concepts you've been learning about still apply
  - C++ automagic replaces some of the manual drudgery

C++ benefits

C	C++
You must call free() yourself to deallocate heap objects.	Language helpfully frees heap objects when owner goes out of scope.
Need a unique name for every function.	Can overload function for different argument types.
Operators like + and == work only for built-in types.	You can overload operators for user-defined types.

C++ downsides

C	C++	
You know exactly when things are freed.	You must call free() yourself to deallocate heap objects.	Language helpfully frees heap objects when owner goes out of scope.	Things get freed when you might not expect it.
You always know what function you are calling.	Need a unique name for every function.	Can overload function for different argument types.	Must know argument types to determine which function gets called.
You know that / means "divide".	Operators like + and == work only for built-in types.	You can overload operators for user-defined types.	You know that `operator/()` takes two arguments.

C++ Versions

- C++ is a little less one language and more multiple iterations of a language
  - Where nothing old every leaves, only new things get added
  - “Within C++, there is a much smaller and cleaner language struggling to get out.” – Bjarne Stoustrop

- One major change was C++11 (2011) which introduced a better method for handling dynamic memory
  - We’ll be using C++14 which has some quality-of-life improvements to that

  - C++17 and C++20 also exist!
    - But don’t add much that we need

Outline

- Why C++?
- **Simple C++ I/O**
- Pass-by-reference
- Vectors

Hello world in C++

src/hello_world.cxx

```cpp
#include <iostream>

int main() {
   std::cout << "Hello World\n";
   return 0;
}
```

The standard C headers are renamed

- Every C header loses the `.h` and gets a `c` added to the front

C version of headers	C++ version of headers

`#include <ctype.h>`
`#include <math.h>`
`#include <stdio.h>`
`#include <string.h>`
	`#include <cctype>`
	`#include <cmath>`
	`#include <cstdio>`
	`#include <cstring>`

The standard C headers are renamed

- Every C header loses the `.h` and gets a `c` added to the front

C version of headers	C++ version of headers

`#include <ctype.h>`
`#include <math.h>`
`#include <stdio.h>`
`#include <string.h>`
	`#include <cctype>`
	`#include <cmath>`
	~~`#include <cstdio>`~~
	~~`#include <cstring>`~~

- And new headers support the similar functionality in a C++ way

```
#include <iostream>
#include <string>
```

You'll use these instead of the C versions because they are easier and safer to use.

More complicated I/O input example

src/io_example.cxx

```
#include <iostream >
int main() {
  std::cout << "Enter a number to square:\n";
  double x;
  std::cin >> x;
  if (!std::cin) {
    std::cerr << "Error: could not read number!\n";
    return 1;
  }
  std::cout << x << " * " << x << " == " << x * x << "\n";
  return 0;
}
```

More complicated I/O input example

New library for I/O

```
#include <iostream >
int main() {
  std::cout << "Enter a number to square:\n";
  double x;
  std::cin >> x;
  if (!std::cin) {
    std::cerr << "Error: could not read number!\n";
    return 1;
  }
  std::cout << x << " * " << x << " == " << x * x << "\n";
  return 0;
}
```

More complicated I/O input example

```
#include <iostream >
int main() {
  std::cout << "Enter a number to square:\n";
  double x;
  std::cin >> x;
  if (!std::cin) {
    std::cerr << "Error: could not read number!\n";
    return 1;
  }
  std::cout << x << " * " << x << " == " << x * x << "\n";
  return 0;
}
```

`main()` and `main(void)` are equivalent

Could still get input argc and argv if wanted

More complicated I/O input example

C++ standard library is in the `std` namespace

```
#include <iostream >
int main() {
  std::cout << "Enter a number to square:\n";
  double x;
  std::cin >> x;
  if (!std::cin) {
    std::cerr << "Error: could not read number!\n";
    return 1;
  }
  std::cout << x << " * " << x << " == " << x * x << "\n";
  return 0;
}
```

More complicated I/O input example

```
#include <iostream >
int main() {
  std::cout << "Enter a number to square:\n";
  double x;
  std::cin >> x;
  if (!std::cin) {
    std::cerr << "Error: could not read number!\n";
    return 1;
  }
  std::cout << x << " * " << x << " == " << x * x << "\n";
  return 0;
}
```

Stream insertion operator writes a value to an output stream

More complicated I/O input example

Stream extraction operator reads from the input stream into an object

```
#include <iostream >
int main() {
  std::cout << "Enter a number to square:\n";

  double x;
  std::cin >> x;
  if (!std::cin) {
    std::cerr << "Error: could not read number!\n";
    return 1;
  }

  std::cout << x << " * " << x << " == " << x * x << "\n";
  return 0;
}
```

More complicated I/O input example

To detect I/O error on a stream, test the stream as if it were a `bool`.

```
#include <iostream >
int main() {
  std::cout << "Enter a number to square:\n";
  double x;
  std::cin >> x;
  if (!std::cin) {
    std::cerr << "Error: could not read number!\n";
    return 1;
  }
  std::cout << x << " * " << x << " == " << x * x << "\n";
  return 0;
}
```

More complicated I/O input example

Stream operators are left-associative and return their left operand

```
#include <iostream >
int main() {
  std::cout << "Enter a number to square:\n";
  double x;
  std::cin >> x;
  if (!std::cin) {
    std::cerr << "Error: could not read number!\n";
    return 1;
  }
  std::cout << x << " * " << x << " == " << x * x << "\n";
  return 0;
}
```

Stream operator chaining

This:

```
std::cout << x << " * " << x << " == " << x * x << "\n";
```

Is equivalent to this:

```
((((std::cout << x) << " * “) << x) << " == “) << x * x) << "\n";
```

Is equivalent to this:

```
std::cout << x;
std::cout << “ * “;
std::cout << x;
std::cout << “ == “;
std::cout << x * x;
std::cout << “\n”;
```

iostream library

- Provides input/output *streams*
  - Sources that you can write characters to or read characters from
  - Same idea as a `FILE*` in C

`std::cin` - standard in
`std::cout` - standard out
`std::cerr` - standard error

- Simple I/O
  - Write using `<<` operator (stream insertion)
  - Read using `>>` operator (stream extraction)

Namespaces in C++

- Namespaces provide additional naming to functions/variables
  - Prevent C problem of "no two functions can have the same name"
  - Refer to `name` as `namespace::name`
  - Defaults to global namespace (just `::name` which is the same as `name`)

  - Basically what we were doing in C anyways
    - `vc_create()`,`ballot_create()`,`ballot_box_create()`

- Avoid `using namespace std;`
  - Eliminates the need to use `std::` for library calls!
  - But also means you must never duplicate a library function name
    - Back to the same problem C had!

Break + Open Question

How does this code know you want a double?

```
double x;

std::cin >> x;
```

Break + Open Question

How does this code know you want a double?

```
double x;

std::cin >> x;
```

Operator overloading!

- You can redefine the meaning of operators in C++
- So `operator>>(istream, double)` is defined to read in a double

- We'll talk more about this in a future lecture

Outline

- Why C++?
- Simple C++ I/O
- **Pass-by-reference**
- Vectors

In C, all arguments are passed as *values*

```
void f(int x, int* p) { ...
```

- In C, every variable names its own object:
  - `x` names 4 bytes capable of containing an `int`
  - `p` names 8 bytes capable of holding the memory address of an `int`

- C allows you to access other objects with pointers
  - But you are still passing a value into the function (a pointer value)

C++ has pass-by-reference

```
void f(int x, int* p , int& r) { ...
```

- `x` and `p` work the same as in C programs
- `r` refers to some other existing `int` object
  - `r` is an alternative *name* for whatever *object* was passed in
  - `r` is borrowed and cannot be `nullptr`
- Use `r` like an ordinary `int` – no need to dereference

C++ reference example: increment

test/reference_examples.cxx

Our C++ testing framework. Similar to how it worked in C!

```
#include <211.h>

void inc_ptr(int* p) {
  *p += 1;
}

void c_style_test(void) {
  int x = 0;
  inc_ptr(&x);
  CHECK_INT( x, 1 );
}
```

```
#include <catch.hxx>

void inc_ref(int& r){
  r += 1;
}

TEST_CASE("C++-style"){
  int x = 0;
  inc_ref(x);
  CHECK( x == 1 );
}
```

Visual representation of references

test/reference_examples.cxx

```
#include <catch.hxx>

void inc_ref(int& r){
   r += 1;
}

TEST_CASE("C++-style"){
   int x = 0;
   inc_ref(x);
   CHECK( x == 1 );
}
```

x: 0

Visual representation of references

test/reference_examples.cxx

```
#include <catch.hxx>

void inc_ref(int& r){
   r += 1;
}

TEST_CASE("C++-style"){
   int x = 0;
   inc_ref(x);
   CHECK( x == 1 );
}
```

Visual representation of references

test/reference_examples.cxx

```
#include <catch.hxx>

void inc_ref(int& r){
  r += 1;
}

TEST_CASE("C++-style"){
  int x = 0;
  inc_ref(x);
  CHECK( x == 1 );
}
```

Same object that was previously named `x`

r: 0

Visual representation of references

test/reference_examples.cxx

```
#include <catch.hxx>

void inc_ref(int& r){
  r += 1;
}

TEST_CASE("C++-style"){
  int x = 0;
  inc_ref(x);
  CHECK( x == 1 );
}
```

r: 1

Visual representation of references

test/reference_examples.cxx

```
#include <catch.hxx>

void inc_ref(int& r){
   r += 1;
}

TEST_CASE("C++-style"){
   int x = 0;
   inc_ref(x);
   CHECK( x == 1 );
}
```

Back here, the object is still named `x`

Swap with references in C++

test/reference_examples.cxx

```
void swap_ref(int& r, int& s) {
  int temp = r;
  r = s;
  s = temp;
}
```

```
TEST_CASE("C++-style swap"){
  int x = 3;
  int y = 4;
  swap_ref(x, y);
  CHECK( x == 4 );
  CHECK( y == 3 );
}
```

Swap with references in C++

test/reference_examples.cxx

```
void swap_ref(int& r, int& s) {
  int temp = r;
  r = s;
  s = temp;
}
```

```
TEST_CASE("C++-style swap"){
  int x = 3;
  int y = 4;
  swap_ref(x, y);
  CHECK( x == 4 );
  CHECK( y == 3 );
}
```

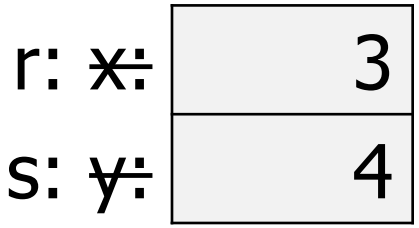
x:	3
y:	4

Swap with references in C++

test/reference_examples.cxx

```
void swap_ref(int& r, int& s) {
  int temp = r;
  r = s;
  s = temp;
}
```

```
TEST_CASE("C++-style swap"){
  int x = 3;
  int y = 4;
  swap_ref(x, y);
  CHECK( x == 4 );
  CHECK( y == 3 );
}
```



Swap with references in C++

test/reference_examples.cxx

```
void swap_ref(int& r, int& s) {
  int temp = r;
  r = s;
  s = temp;
}
```

```
TEST_CASE("C++-style swap"){
  int x = 3;
  int y = 4;
  swap_ref(x, y);
  CHECK( x == 4 );
  CHECK( y == 3 );
}
```

r: ~~x:~~	3
s: ~~y:~~	4
temp:	3

Swap with references in C++

test/reference_examples.cxx

```
void swap_ref(int& r, int& s) {
  int temp = r;
  r = s;
  s = temp;
}
```

```
TEST_CASE("C++-style swap"){
  int x = 3;
  int y = 4;
  swap_ref(x, y);
  CHECK( x == 4 );
  CHECK( y == 3 );
}
```

r: ~~x:~~	4
s: ~~y:~~	4
temp:	3

Swap with references in C++

test/reference_examples.cxx

```
void swap_ref(int& r, int& s) {
  int temp = r;
  r = s;
  s = temp;
}
```

```
TEST_CASE("C++-style swap"){
  int x = 3;
  int y = 4;
  swap_ref(x, y);
  CHECK( x == 4 );
  CHECK( y == 3 );
}
```

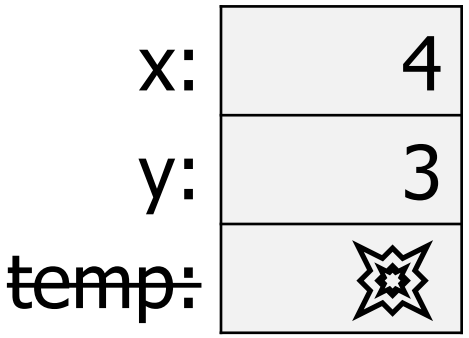
r: ~~x:~~	4
s: ~~y:~~	3
temp:	3

Swap with references in C++

test/reference_examples.cxx

```
void swap_ref(int& r, int& s) {
  int temp = r;
  r = s;
  s = temp;
}
```

```
TEST_CASE("C++-style swap"){
  int x = 3;
  int y = 4;
  swap_ref(x, y);
  CHECK( x == 4 );
  CHECK( y == 3 );
}
```



References can be thought of as “syntactic sugar”

1. Replace every declared references with a pointer
2. Dereference each use of the variable
3. Take pointer of each variable passed in

```
void swap(int& r, int& s)
{
  int temp = r;
  r = s;
  s = temp;
}

swap(x, v[3]);
```

```
void swap(int* rp, int* sp)
{
  int temp = *rp;
  *rp = *sp;
  *sp = temp;
}

swap(&x, &v[3]);
```

References can be thought of as "syntactic sugar"

1. Replace every declared references with a pointer
2. Dereference each use of the variable
3. Take pointer of each variable passed in

```
void swap(int& r, int& s)
{
  int temp = r;
  r = s;
  s = temp;
}

swap(x, v[3]);
```

```
void swap(int* rp, int* sp)
{
  int temp = *rp;
  *rp = *sp;
  *sp = temp;
}

swap(&x, &v[3]);
```

References can be thought of as "syntactic sugar"

1. Replace every declared references with a pointer
2. Dereference each use of the variable
3. Take pointer of each variable passed in

```
void swap(int& r, int& s)
{
  int temp = r;
  r = s;
  s = temp;
}

swap(x, v[3]);
```

```
void swap(int* rp, int* sp)
{
  int temp = *rp;
  *rp = *sp;
  *sp = temp;
}

swap(&x, &v[3]);
```

References can be thought of as "syntactic sugar"

1. Replace every declared references with a pointer
2. Dereference each use of the variable
3. Take pointer of each variable passed in

```
void swap(int& r, int& s)
{
  int temp = r;
  r = s;
  s = temp;
}

swap(x, v[3]);
```

```
void swap(int* rp, int* sp)
{
  int temp = *rp;
  *rp = *sp;
  *sp = temp;
}

swap(&x, &v[3]);
```

This “desugaring” approach can explain more complicated references

References version

```
entry& e = entries[i];
std::string const& n = e.name;

if (n == current) {
  ++e.count;
}
```

“Desugared” pointer version

```
entry* pe = &(entries[i]);
std::string const* pn = &(pe->name);

if (*pn == current) {
  ++pe->count;
//++(*pe).count
}
```

- Note: `std::string` types can be compared with `==`
  - Prefer `std::string` over `char*` in C++

Break + Question: Does this swap work?

```
void alt_swap(int& r, int& s)
{
  int& temp = r;
  r = s;
  s = temp;
}
```

Break + Question: Does this swap work?

```
void alt_swap(int& r, int& s)
{
  int& temp = r;
  r = s;
  s = temp;
}
```

r:	4
s:	3

Break + Question: Does this swap work?

```
void alt_swap(int& r, int& s)
{
  int& temp = r;
  r = s;
  s = temp;
}
```

`r` and `temp` both name the same object!

Break + Question: Does this swap work?

```
void alt_swap(int& r, int& s)
{
  int& temp = r;
  r = s;
  s = temp;
}
```

`r` and `temp` both name the same object!

Break + Question: Does this swap work?

```
void alt_swap(int& r, int& s)
{
  int& temp = r;
  r = s;
  s = temp;
}
```

`r` and `temp` both name the same object!

This version of swap is broken!

Break + Question: Does this swap work?

References version

```
void alt_swap(int& r, int& s)
{
  int& temp = r;
  r = s;
  s = temp;
}
```

“Desugared” pointer version

```
void alt_swap(int* rp, int* sp)
{
  int* tempp = &*rp;
  *rp = *sp;
  *sp = *tempp;
}
```

Outline

- Why C++?
- Simple C++ I/O
- Pass-by-reference
- **Vectors**

C++ libraries provide various useful structures for you

- C libraries had some functions that would let you interact with things like files or the user

- C++ has those, but also has libraries with data structures and with various algorithms (such as sorting)
  - C++ data structures (containers): https://cplusplus.com/reference/stl/
  - C++ algorithms: https://cplusplus.com/reference/algorithm/

C++ Vectors

- One example C++ library: Vector
  - An automatically expanding "array" capable of holding any type
  - `std::vector<TYPE>` to choose what type it should hold
    - `std::vector<int>`, `std::vector<double>`, etc.
    - This idea is known as "generics". We'll discuss in a later lecture

- Creating a vector (there are many ways)

```
std::vector<TYPE> myvector(); //empty vector of with no size
std::vector<TYPE> myvector(len); //vector of size len with uninitialized values
std::vector<TYPE> myvector(len, val); //vector of size len with values set to val

std::vector<TYPE> myvector{val1, val2, val3, ...};
//vector with initial values, set to a size to hold them all
```

Other useful Vector operations

- `vec[n]` is used to get the value at index `n`
  - Works just like a C array
  - Still **UNDEFINED BEHAVIOR** if `n` is out of bounds for the Vector

- `vec.at(n)` accesses value at index `n`
  - Just like square brackets, but throws an exception if out-of-bounds
  - Exceptions: new way of signaling errors. Will talk about in later lecture

- `vec.size()` returns the length of the Vector

- `vec.push_back()` and `vec.pop_back()` add/remove items
  - And resize the Vector automatically as needed

Example vector code

test/vector_examples.cxx

- Play around with vectors

C++ allows for simpler iteration (like Python)

```
double sum_vec(std::vector<double> const& vec){
  double result = 0;
  for (double val : vec) {
    result += val;
  }
  return result;
}
```

Iterates over elements in the vector, not indices!

Modifying elements inside the vector

- Warning: make sure you're modifying the actual vector element

```
void dec_vec_wrong(std::vector<int> &vec){
  for (int val : vec){
    --val;
  }
}
```

Each `val` is a copy of the value in the vector

Modifying elements inside the vector

- Warning: make sure you're modifying the actual vector element

```
void dec_vec_wrong(std::vector<int> &vec){
  for (int val : vec){
    --val;
  }
}
```

Each `val` is a copy of the value in the vector

```
void dec_vec_right(std::vector<int> &vec){
  for (int& val : vec){
    --val;
  }
}
```

Each `val` is a reference to the value in the vector.

So modifying it works!

Outline

- Why C++?
- Simple C++ I/O
- Pass-by-reference
- Vectors