C++ Intro
CSE 333

Instructor: Hannah C. Tang

Teaching Assistants:

Deeksha Vatwani	Hannah Jiang	Jen Xu
Justin Tysdal	Leanna Nguyen	Sayuj Shahi
Wei Wu	Yiqing Wang	Youssef Ben Taleb

Today's Goals

❖ An introduction to C++

- Some comparisons to C and shortcomings that C++ addresses
- Give you a perspective on how to learn C++
- Kick the tires and look at some code

❖ **Advice:** You *must* read related sections in the *C++ Primer*

- It's hard to learn the "why is it done this way" from reference docs, and even harder to learn from random stuff on the web
- Lectures and examples will introduce the main ideas, but aren't everything you'll ~~want~~ need to understand
- 3 hours of web searching *might* save you 20 min. of reading in the *Primer* – but is that a good tradeoff?
- And *free* access through UW libraries (O'Reilly books online)

C

❖ We had to work hard to mimic encapsulation, abstraction

- **Encapsulation:** hiding implementation details
  - Used header file conventions and the "static" specifier to separate private functions from public functions
  - Cast structures to (void*) to hide implementation-specific details
- **Abstraction:** associating behavior with encapsulated state
  - Function that operate on a LinkedList were not really tied to the linked list structure
  - We passed a linked list to a function, rather than invoking a method on a linked list instance

✨ C++ ✨
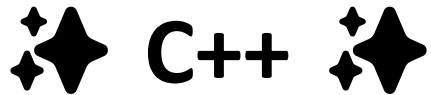
- A major addition is support for classes and objects!
  - Classes
    - Public, private, and protected **methods** and **instance variables**
    - (multiple!) inheritance
  - Polymorphism
    - Static polymorphism: multiple functions or methods with the same name, but different argument types (overloading)
      - Works for all functions, not just class members
    - Dynamic (subtype) polymorphism: derived classes can override methods of parents, and methods will be dispatched correctly

# C

❖ **We had to emulate generic data structures**

- Generic linked list using `void*` payload

- Pass function pointers to generalize different "methods" for data structures

  - Comparisons, deallocation, pickling up state, etc.

✦ C++ ✦
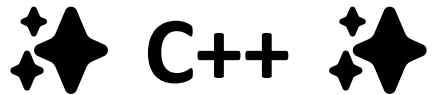
❖ Supports templates to facilitate generic data types

- Parametric polymorphism – same idea as Java generics, but different in details, particularly implementation

- To declare that x is a vector of ints: `vector<int> x;`

- To declare that x is a vector of strings: `vector<string> x;`

- To declare that x is a vector of (vectors of floats):
  `vector<vector<float>> x;`

C

❖ **We had to be careful about namespace collisions**

▪ C distinguishes between external and internal linkage

- Use `static` to prevent a name from being visible outside a source file (as close as C gets to "private")

- Otherwise, name is global and visible everywhere

▪ We used naming conventions to help avoid collisions in the global namespace

- *e.g.* **`LL`**`IteratorNext` vs. **`HT`**`IteratorNext`, etc.

✦ C++ ✦
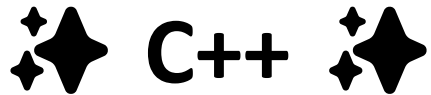
❖ Permits a module to define its own namespace!

■ The linked list module could define an "`LL`" namespace while the hash table module could define an "`HT`" namespace

■ Both modules could define an Iterator class

• One would be globally named `LL::Iterator`

• The other would be globally named `HT::Iterator`

❖ Classes also allow duplicate names without collisions

■ Namespaces group and isolate names in collections of classes and other "global" things (somewhat like Java packages)

• Entire C++ standard library is in a namespace `std` (more later…)

C

❖ **C does not provide any standard data structures**

- ■ We had to implement our own linked list and hash table

- ■ As a C programmer, you often reinvent the wheel… poorly

  - • Maybe if you're clever you'll use somebody else's libraries

  - • But C's lack of abstraction, encapsulation, and generics means you'll probably end up tinkering with them or tweak your code to use them

✨ C++ ✨
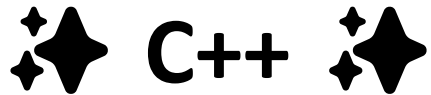
❖ The C++ standard library is huge!

- **Generic containers:** bitset, queue, list, associative array (including hash table), deque, set, stack, and vector
  - And iterators for most of these
- **A `string` class:** hides the implementation of strings
- **Streams:** allows you to stream data to and from objects, consoles, files, strings, and so on
- And more…

C

❖ **Error handling is a pain**

  ▪ Have to define error codes and return them

  ▪ Customers have to understand error code conventions and need to constantly test return values

  ▪ *e.g.* if `a()` calls `b()`, which calls `c()`

    • `a` depends on `b` to propagate an error in `c` back to it

✨ C++ ✨

❖ **Error handling is STILL a pain, but now we have exceptions**

   ▪ `try` / `throw` / `catch`

   ▪ **If used with discipline, can simplify error processing**

      • But, if used carelessly, can complicate memory management

      • Consider: `a()` calls `b()`, which calls `c()`

         – If `c()` throws an exception that `b()` doesn't catch, you might not get a chance to clean up resources allocated inside `b()`

   ▪ **But much C++ code still needs to work with C & old C++ libraries that are not exception-safe, so still uses return codes, exit(), etc.**

      • We won't use (and Google style guide doesn't use either)

Some Tasks Still Hurt in ✨ C++ ✨

❖ **Memory management**

- C++ has no garbage collector
  - You have to manage memory allocation and deallocation and track ownership of memory
  - It's still possible to have leaks, double frees, and so on
- But there are some things that help
  - "Smart pointers"
    - Classes that encapsulate pointers and track reference counts
    - Deallocate memory when the reference count goes to zero
  - C++'s destructors permit a pattern known as "Resource Allocation Is Initialization" (RAII) (terrible name but super useful idea)
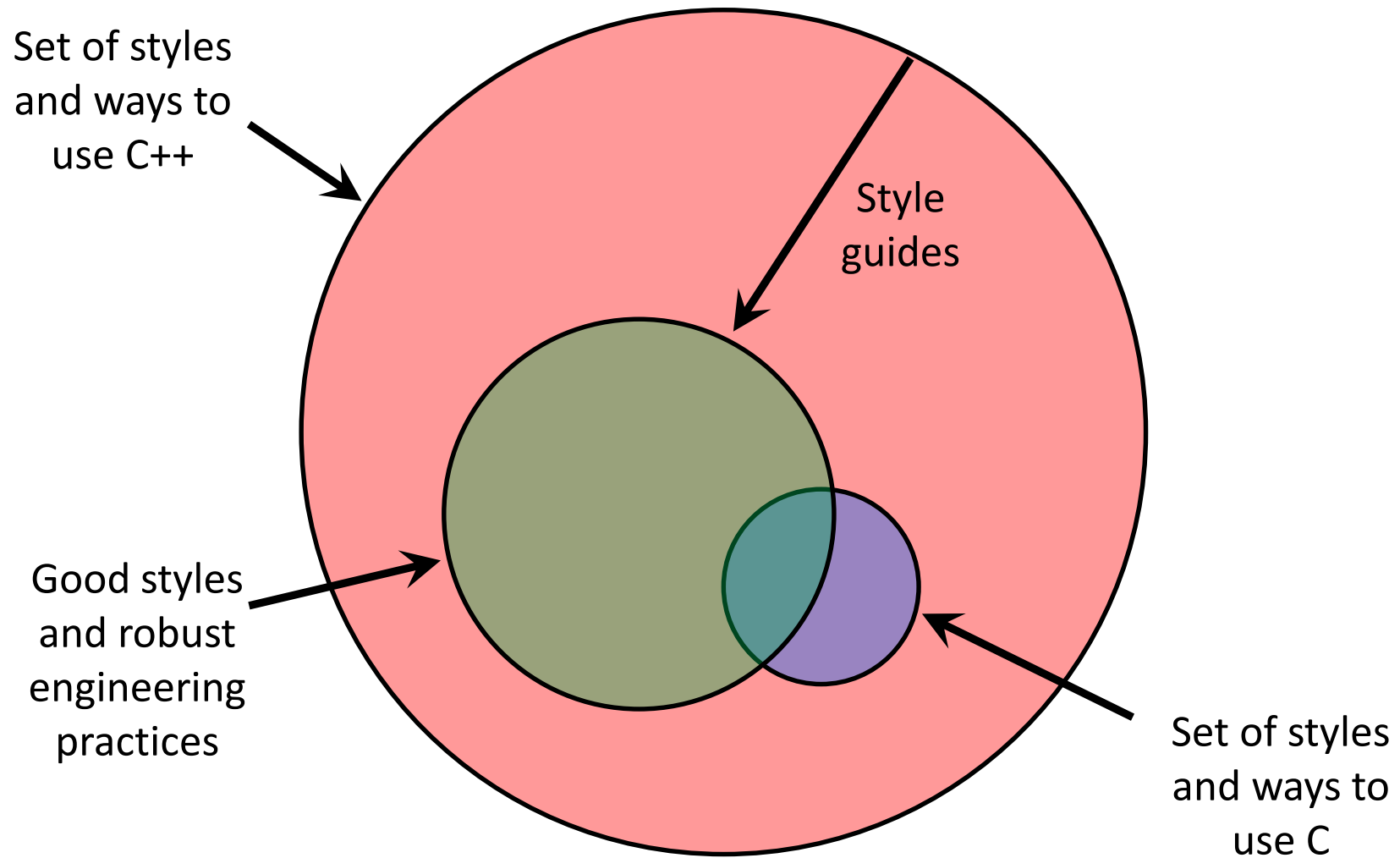   - Useful for releasing memory, locks, database transactions, and more

Some Tasks Still Hurt in ✦ C++ ✦

❖ C++ doesn't guarantee type or memory safety

▪ You can still:

- Forcibly cast pointers between incompatible types

- Walk off the end of an array and smash memory

- Have dangling pointers

- Conjure up a pointer to an arbitrary address of your choosing

✨ C++ ✨ Has Many, Many Features

❖ Operator overloading

- ▪ Your class can define methods for handling "+", "–>", etc.

❖ Object constructors, destructors

- ▪ Particularly handy for stack-allocated objects

❖ Reference types

- ▪ True call-by-reference instead of always call-by-value

❖ Advanced Objects

- ▪ Multiple inheritance, virtual base classes, dynamic dispatch

How to Think About ✨ C++ ✨

Set of styles and ways to use C++

Style guides

Good styles and robust engineering practices

Set of styles and ways to use C

Or…

In the hands of a disciplined programmer, C++ is a powerful tool

But if you're not so disciplined about how you use C++…

Hello World in C

helloworld.c

```c
#include <stdio.h>    // for printf()
#include <stdlib.h>   // for EXIT_SUCCESS

int main(int argc, char** argv) {
  printf("Hello, World!\n");
  return EXIT_SUCCESS;
}
```

❖ You never had a chance to write this!

- Compile with `gcc`:

```
gcc -Wall -g -std=c17 -o hello helloworld.c
```

- You should be able to describe in detail everything in this code

Hello World in ✦ C++ ✦

helloworld.cc

```cpp
#include <iostream>  // for cout, endl
#include <cstdlib>   // for EXIT_SUCCESS

int main(int argc, char** argv) {
  std::cout << "Hello, World!" << std::endl;
  return EXIT_SUCCESS;
}
```

❖ Looks simple enough…

- Compile with `g++` instead of `gcc`:

```
g++ -Wall -g -std=c++17 -o helloworld helloworld.cc
```

- Let's walk through the program step-by-step to highlight some differences

Poll Everywhere

pollev.com/uwcse333

❖ Let's play "Spot the Difference"!

Spot 10 differences

helloworld.c

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  printf("Hello, World!\n");

  return EXIT_SUCCESS;
}
```

helloworld.cc

```cpp
#include <iostream>
#include <cstdlib>

int main(int argc, char** argv) {
  std::cout << "Hello, World!"
            << std::endl;
  return EXIT_SUCCESS;
}
```

Hello World in C++

helloworld.cc

```cpp
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS

int main(int argc, char** argv) {
  std::cout << "Hello, World!" << std::endl;
  return EXIT_SUCCESS;
}
```

❖ `iostream` is part of the **C++** standard library

- Note: you don't write "`.h`" when you include C++ standard library headers
  - But you *do* for local headers (*e.g.* `#include "ll.h"`)
- `iostream` declares stream *object* instances in the "`std`" namespace
  - *e.g.* `std::cin, std::cout, std::cerr`

Hello World in C++

helloworld.cc

```cpp
#include <iostream>  // for cout, endl
#include <cstdlib>   // for EXIT_SUCCESS

int main(int argc, char** argv) {
  std::cout << "Hello, World!" << std::endl;
  return EXIT_SUCCESS;
}
```

❖ `cstdlib` is the **C** standard library's `stdlib.h`

- Nearly all C standard library functions are available to you

  - For C header `foo.h`, you should `#include <cfoo>`

- We include it here for `EXIT_SUCCESS`, as usual

Hello World in C++

helloworld.cc

```cpp
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS

int main(int argc, char** argv) {
  std::cout << "Hello, World!" << std::endl;
  return EXIT_SUCCESS;
}
```

❖ `std::cout` is the "`cout`" object instance declared by `iostream`, living within the "`std`" namespace

- ▪ C++'s name for `stdout`

- ▪ `std:cout` is an object of class `ostream`
  - http://www.cplusplus.com/reference/ostream/ostream/

- ▪ Used to format and write output to the console

- ▪ The entire standard library is in the namespace `std`

Hello World in C++

helloworld.cc

```cpp
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS

int main(int argc, char** argv) {
  std::cout << "Hello, World!" << std::endl;
  return EXIT_SUCCESS;
}
```

❖ C++ distinguishes between objects and primitive types

- These include the familiar ones from C:

  `char`, `short`, `int`, `long`, `float`, `double`, etc.

- C++ also defines `bool` as a primitive type (woo-hoo!)

  - Use it!

  - (but bool and int values silently convert types for compatiblity)

Hello World in C++

helloworld.cc

```cpp
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS

int main(int argc, char** argv) {
  std::cout << "Hello, World!" << std::endl;
  return EXIT_SUCCESS;
}
```

Vector v1
v2
v1+v2

❖ "<<" is an operator defined by the C++ language

   ▪ Defined in C as well: usually it bit-shifts integers (in C/C++)

   ▪ C++ allows classes and functions to overload operators!

      • Here, the `ostream` class overloads "<<"

      • *i.e.* it defines different member functions (methods) that are invoked
        when an `ostream` is the left-hand side of the << operator

Hello World in C++

helloworld.cc

```cpp
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS

int main(int argc, char** argv) {
  std::cout << "Hello, World!" << std::endl;
  return EXIT_SUCCESS;
}
```

- ❖ `ostream` has many different methods to handle `<<`
  - The functions differ in the type of the right-hand side (RHS) of `<<`
  - *e.g.* if you do `std::cout << "foo";` then C++ invokes `cout`'s function to handle `<<` with RHS `char*`

Hello World in C++

helloworld.cc

```cpp
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS

int main(int argc, char** argv) {
  std::cout << "Hello, World!" << std::endl;
  return EXIT_SUCCESS;
}
```

❖ The `ostream` class' member functions that handle `<<` return a reference to themselves

  ▪ When `std::cout << "Hello, World!";` is evaluated:

    • A member function of the `std::cout` object is invoked

    • It buffers the string `"Hello, World!"` for the console

    • And it returns a reference to `std::cout`

Hello World in C++

helloworld.cc

```cpp
#include <iostream>  // for cout, endl
#include <cstdlib>   // for EXIT_SUCCESS

int main(int argc, char** argv) {
  std::cout << "Hello, World!" << std::endl;
  return EXIT_SUCCESS;
}
```

❖ Next, another member function on `std::cout` is invoked to handle `<<` with RHS `std::endl`

- `std::endl` is a pointer to a "manipulator" function
  - This manipulator function writes newline (`'\n'`) to the `ostream` it is invoked on and then flushes the `ostream`'s buffer
  - This *enforces* that something is printed to the console at this point

Wow…

helloworld.cc

```cpp
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS

int main(int argc, char** argv) {
  std::cout << "Hello, World!" << std::endl;
  return EXIT_SUCCESS;
}
```

❖ You should be surprised and scared at this point

■ C++ makes it easy to hide a significant amount of complexity

• It's powerful, but really dangerous

• Once you mix everything together (templates, operator overloading, method overloading, generics, multiple inheritance), it can get *really* hard to know what's actually happening!

Let's Refine It a Bit

helloworld2.cc

```cpp
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS
#include <string>

using namespace std;

int main(int argc, char** argv) {
  string hello("Hello, World!");
  cout << hello << endl;
  return EXIT_SUCCESS;
}
```

❖ C++'s standard library has a `std::string` class

- Include the `string` header to use it
  - Seems to be automatically included in `iostream` on CSE Linux environment (C++17) – but include it explicitly anyway if you use it
- http://www.cplusplus.com/reference/string/

Let's Refine It a Bit

helloworld2.cc

```cpp
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS
#include <string>

using namespace std;

int main(int argc, char** argv) {
  string hello("Hello, World!");
  cout << hello << endl;
  return EXIT_SUCCESS;
}
```

> Google style guide says don't do this, but for CSE333, `using namespace std` is ok in .cc (never in .h)

❖ The `using` keyword introduces a namespace (or part of) into the current region

- ▪ `using namespace std;` imports all names from `std::`
- ▪ `using std::cout;` imports *only* `std::cout` (used as `cout`)

Let's Refine It a Bit

helloworld2.cc

```cpp
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS
#include <string>

using namespace std;

int main(int argc, char** argv) {
  string hello("Hello, World!");
  cout << hello << endl;
  return EXIT_SUCCESS;
}
```

❖ Benefits of `using namespace std;`

- We can now refer to `std::string` as `string`, `std::cout` as `cout`, and `std::endl` as `endl`
  - Google style guide says never use `using namespace`, only `using` for individual items; but for 333 `using namespace std` is ok

Let's Refine It a Bit

helloworld2.cc

```cpp
#include <iostream>  // for cout, endl
#include <cstdlib>   // for EXIT_SUCCESS
#include <string>

using namespace std;

int main(int argc, char** argv) {
  string hello("Hello, World!");
  cout << hello << endl;
  return EXIT_SUCCESS;
}
```

❖ **Here we are instantiating a `std::string` object *on the stack* (an ordinary local variable)**

- Passing the C string `"Hello, World!"` to its constructor method

- `hello` is deallocated (and its destructor invoked) when `main` returns

Let's Refine It a Bit

helloworld2.cc

```cpp
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS
#include <string>

using namespace std;

int main(int argc, char** argv) {
  string hello("Hello, World!");
  cout << hello << endl;
  return EXIT_SUCCESS;
}
```

❖ The C++ string library also overloads the << operator

▪ Defines a function (*not* an object method) that is invoked when the LHS is `ostream` and the RHS is `std::string`

• http://www.cplusplus.com/reference/string/string/operator<</

String Concatenation

concat.cc

```cpp
#include <iostream>  // for cout, endl
#include <cstdlib>   // for EXIT_SUCCESS
#include <string>

using namespace std;

int main(int argc, char** argv) {
  string hello("Hello");
  hello = hello + ", World!";
  cout << hello << endl;
  return EXIT_SUCCESS;
}
```

❖ The string class overloads the "+" operator
  ▪ Creates and returns a new string that is the concatenation of the
    LHS and RHS

String Assignment

concat.cc

```cpp
#include <iostream>  // for cout, endl
#include <cstdlib>   // for EXIT_SUCCESS
#include <string>

using namespace std;

int main(int argc, char** argv) {
  string hello("Hello");
  hello = hello + ", World!";
  cout << hello << endl;
  return EXIT_SUCCESS;
}
```

❖ The string class overloads the "=" operator

  ▪ Copies the RHS and replaces the string's contents with it

String Manipulation

concat.cc

```cpp
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS
#include <string>

using namespace std;

int main(int argc, char** argv) {
  string hello("Hello");
  hello = hello + ", World!";
  cout << hello << endl;
  return EXIT_SUCCESS;
}
```

❖ This statement is complex!

- First "+" creates a string that is the concatenation of `hello`'s current contents and `", World!"`

- Then "=" creates a copy of the concatenation to store in `hello`

- Without the syntactic sugar:

  - `hello.operator=(hello.operator+(", World!"));`

41

Stream Manipulators

```cpp
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS
#include <iomanip>    // for setw, hex, dec

using namespace std;

int main(int argc, char** argv) {
  cout << "Hi! " << setw(4) << 5 << " " << 5 << endl;
  cout << hex << 16 << " " << 13 << endl;
  cout << dec << 16 << " " << 13 << endl;
  return EXIT_SUCCESS;
}
```

❖ `iomanip` defines a set of stream manipulator functions

- Pass them to a stream to affect formatting

  - http://www.cplusplus.com/reference/iomanip/

  - http://www.cplusplus.com/reference/ios/

Stream Manipulators

manip.cc

```cpp
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS
#include <iomanip>    // for setw, hex, dec

using namespace std;

int main(int argc, char** argv) {
  cout << "Hi! " << setw(4) << 5 << " " << 5 << endl;
  cout << hex << 16 << " " << 13 << endl;
  cout << dec << 16 << " " << 13 << endl;
  return EXIT_SUCCESS;
}
```

❖ `setw(x)` sets the width of the *next* field to `x`

- Only affects the next thing sent to the output stream (*i.e.* it is not persistent)

43

Stream Manipulators

manip.cc

```cpp
#include <iostream>  // for cout, endl
#include <cstdlib>   // for EXIT_SUCCESS
#include <iomanip>   // for setw, hex, dec

using namespace std;

int main(int argc, char** argv) {
  cout << "Hi! " << setw(4) << 5 << " " << 5 << endl;
  cout << hex << 16 << " " << 13 << endl;
  cout << dec << 16 << " " << 13 << endl;
  return EXIT_SUCCESS;
}
```

❖ `hex`, `dec`, and `oct` set the numerical base for *integer* output to the stream

- Stays in effect until you set the stream to another base (*i.e.* it is persistent)

C and C++

helloworld3.cc

```c
#include <cstdio>    // for printf
#include <cstdlib>   // for EXIT_SUCCESS

int main(int argc, char** argv) {
  printf("Hello from C!\n");
  return EXIT_SUCCESS;
}
```

❖ C is (roughly) a subset of C++

- You can still use `printf` – but bad style in ordinary C++ code
- Can mix C and C++ idioms if needed to work with existing code, but avoid mixing if you can
  - Use C++(17)

Reading

echonum.cc

```cpp
#include <iostream>
#include <cstdlib>

using namespace std;

int main(int argc, char** argv) {
  int num;
  cout << "Type a number: ";
  cin >> num;
  cout << "You typed: " << num << endl;
  return EXIT_SUCCESS;
}
```

❖ `std::cin` is an object instance of class `istream`

- Supports the `>>` operator for "extraction"

  - Can be used in conditionals – (`std::cin>>num`) is `true` if successful

- Has a `getline()` method and methods to detect and clear errors

Poll Everywhere

❖ **How many *different* versions of `operator<<` are called?**

  ▪ For now, ignore manipulators like `hex` and `endl`

  ▪ Also, what is output?

msg.cc

```cpp
#include <iostream>
#include <cstdlib>
#include <string>
#include <iomanip>

using namespace std;

int main(int argc, char** argv) {
  int n = 172;
  string str("m");
  str += "y";
  cout << str << hex << setw(2)
       << 15U << n << "e!" << endl;
  return EXIT_SUCCESS;
}
```

A. 1

B. 2

C. 3

D. 4

E. We're lost…

Extra Exercise #1

❖ Write a C++ program that uses stream to:

 ▪ Prompt the user to type 5 floats

 ▪ Prints them out in opposite order with 4 digits of precision