Intro to C++

Computer Systems Programming, Spring 2023

Instructor: Travis McGaha

TAs:

Kevine Bernat
Mati Davis
Chandravaran Kunjeti
Shufan Liu
Jialin Cai
Donglun He
Heyi Liu
Eddy Yang

Logistics

- Pre-Semester Survey: **Due Tuesday 1/24 @ 11:59 PM**
  - Survey to get information on how to make the course better suited to everyone
- HW00: **Due Thursday 1/26 @ 11:59 PM**
  - Implement LinkedList & HashTable
  - You should have everything you need after this lecture
  - HWs can take a while
  - **DO NOT PUT THIS OFF, PLEASE GET STARTED**

Lecture Outline

- C vs C++ (Hello World)
  - I/O stream
  - string
- new/delete
- const

Today's Goals

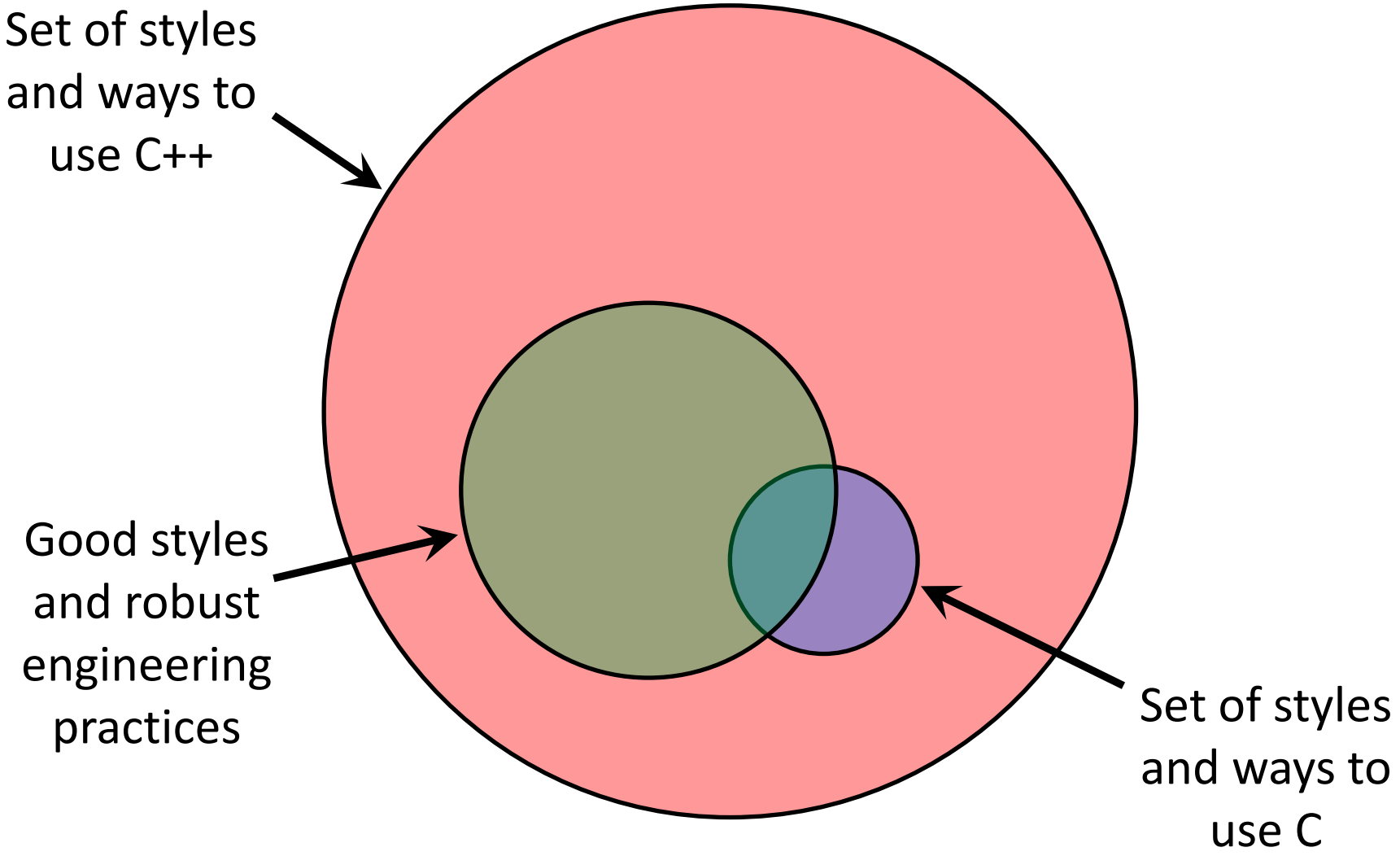
- An introduction to C++
  - Give you a perspective on how to learn C++
  - Compare and contrast some aspects of C vs C++
  - Introduce const

- **Note:**
  - There is a LOT to C++, and we do not have time to go over it all
  - The lectures this week will introduce some basic C++, expect more to occur after spring break
  - HIGHLY recommend you use a C++ reference when coding

C++ Design Philosophy

> Many C++ design decisions have their roots in my dislike for forcing people to do things in some particular way [...] Often, I was tempted to outlaw a feature I personally disliked, I refrained from doing so because **I did not think I had the right to force my views on others**.
>
> *Bjarne Stroustrup. Source: The Design and Evolution of C++ (1.3 General Background)*

How to Think About 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[]) {
  char* str = "Hello, World!\n";
  printf("%s\n", str);
  return EXIT_SUCCESS;
}
```

- You never had a chance to write this (in this class)!
  - Compile with `gcc`:

```
gcc -Wall -g -std=c11 -o hello helloworld.c
```

Hello World in C++

helloworld.cc

```
#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++11 -o helloworld helloworld.cc
```

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

Hello World in C++

helloworld.cc

```
#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 "Deque.h"`)
  - `iostream` declares stream *object* instances in the "`std`" namespace
    - *e.g.* `std::cin`, `std::cout`, `std::cerr`

Hello World in C++

helloworld.cc

```
#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 `math.h`, you should `#include <cmath>`
  - We include it here for `EXIT_SUCCESS`

Hello World in C++

helloworld.cc

```
#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

```
#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!

Hello World in C++

helloworld.cc

```
#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;
}
```

- "<<" 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

```
#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

```
#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

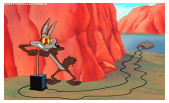
```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS

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

```
#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 (maybe) 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

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

Only doing "**using namespace std**" To save space on slides

helloworld2.cc

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

using namespace std;

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

- 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

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

Let's Refine It a Bit

helloworld2.cc

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

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

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

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

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS
#include <string>      // for 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!"));`

C and C++

helloworld3.cc

```
#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
    - E.g. Use `std::cerr` instead of `fprintf(stderr, …)`
  - Can mix C and C++ idioms if needed to work with existing code, but avoid mixing if you can
    - **Use C++**

C++ Documentation

- As said, there is a LOT to C++
  - There are a lot of functions, objects, features, etc
  - We will NOT have time to talk about them all

- We highly recommend you make use of a C++ reference
  - cplusplus.com
    - Most find this one easier to read
    - Probably has the information you need
  - cppreference.com
    - Much more detailed (in Travis' opinion)
    - Is in various languages (scroll to the bottom)

Lecture Outline

- C vs C++ (Hello World)
  - I/O stream
  - string
- **new/delete**
- const

C++11 `nullptr`

- C and C++ have long used `NULL` as a pointer value that references nothing

- C++11 introduced a new literal for this: `nullptr`
  - New reserved word
  - Interchangeable with `NULL` for all practical purposes, but it has type `T*` for any/every `T`, and is not an integer value
    - Avoids funny edge cases (see C++ references for details)
    - Still can convert to/from integer `0` for tests, assignment, etc.
  - Advice: prefer `nullptr` in C++11 code
    - Though `NULL` will also be around for a long, long time

new/delete

- To allocate on the heap using C++, you use the `new` keyword instead of **`malloc`**`()` from `stdlib.h`
  - You can use new to allocate an object (*e.g.* `new Point`)
  - You can use new to allocate a primitive type (*e.g.* `new int`)

- To deallocate a heap-allocated object or primitive, use the `delete` keyword instead of **`free`**`()` from `stdlib.h`
  - Don't mix and match!
    - *Never* **`free`**`()` something allocated with `new`
    - *Never* `delete` something allocated with **`malloc`**`()`
    - Careful if you're using a legacy C code library or module in C++

new/delete Behavior

- `new` behavior:
  - When allocating you can specify a constructor or initial value
    - (*e.g.* `new Point(1, 2)`) or (*e.g.* `new string("hi")`)
  - If no initialization specified, it will use default constructor for objects, garbage for primitives More on constructors in Wednesday's lecture
  - You don't need to check that `new` returns `nullptr`
    - When an error is encountered, an exception is thrown (that we won't worry about)

- `delete` behavior:
  - If you `delete` already `delete`d memory, then you will get undefined behavior. (Same as when you double **`free`** in c)

new/delete Example

```
int* AllocateInt(int x) {
  int* heapy_int = new int;
  *heapy_int = x;
  return heapy_int;
}
```

```
string* AllocateStr(char* str) {
  string* heapy_str = new string(str);
  return heapy_str;
}
```

heappoint.cc

```
#include "Point.h"

...  // definitions of AllocateInt() and AllocatePoint()

int main() {
  string* x = AllocateStr("Hello 595!");
  int* y = AllocateInt(3);

  cout << "x's value: " << *x << endl;
  cout << "y: " << y << ", *y: " << *y << endl;

  delete x;
  delete y;
  return EXIT_SUCCESS;
}
```

Dynamically Allocated Arrays

new still returns a pointer of specified type

- To dynamically allocate an array:
  - Default initialize: `type* name = new type[size];`

- To dynamically deallocate an array:
  - Use `delete[] name;`
  - It is an *incorrect* to use "`delete name;`" on an array
    - The compiler probably won't catch this, though (!) because it can't always tell if `name*` was allocated with `new type[size];` or `new type;`
      - Especially inside a function where a pointer parameter could point to a single item or an array and there's no way to tell which!
    - Result of wrong `delete` is undefined behavior

Arrays Example (primitive)

arrays.cc

```
#include "Point.h"

int main() {
  int stack_int;  // stack (garbage)
  int* heap_int = new int; // heap (garbage)
  int* heap_int_init = new int(12); // heap (12)

  int stack_arr[3]; // stack (garbage)
  int* heap_arr = new int[3]; // heap (garbage)

  int* heap_arr_init_val = new int[3](); // heap (0,0,0)
  int* heap_arr_init_lst = new int[3]{4, 5};   // C++11
                                               // heap (4,5,0)
  ...

  delete heap_int;              // ok
  delete heap_int_init;         // ok
  delete heap_arr;              // BAD
  delete[] heap_arr_init_val;   // ok

  return EXIT_SUCCESS;
}
```

`malloc` vs. `new`

	`malloc()`	`new`
What is it?	a function	an operator or keyword
How often used (in C)?	often	never
How often used (in C++)?	rarely	often
Allocated memory for	anything	arrays, structs, objects, primitives
Returns	a `void*` (*usually cast*)	appropriate pointer type (*doesn't need a cast*)
When out of memory	returns `NULL`	throws an exception
Deallocating	`free()`	`delete` or `delete[]`

Lecture Outline

- C vs C++ (Hello World)
  - I/O stream
  - string
- **new/delete**
- const

const

- `const`: this cannot be changed/mutated
  - Used *much* more in C++ than in C
  - Signal of intent to compiler; meaningless at hardware level
    - Results in compile-time errors

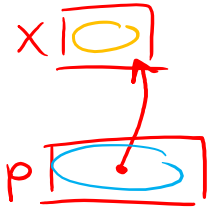
```
void BrokenPrintSquare(const int* i) {
  *i = (*i)*(*i);  // compiler error here!
  std::cout << *i << std::endl;
}

int main(int argc, char** argv) {
  int j = 2;
  BrokenPrintSquare(&j);
  return EXIT_SUCCESS;
}
```

brokenpassbyrefconst.cc

`const` and Pointers

- Pointers can change data in two different contexts:
  1) You can change the value of the pointer
  2) You can change the thing the pointer points to (via dereference)

```
int x;
int *p = &x;
```



- `const` can be used to prevent either/both of these behaviors!
  - `const` next to pointer name means you can't change the value of the pointer

```
int * const p;
```

  - `const` next to data type pointed to means you can't use this pointer to change the thing being pointed to

```
const int * p;
```

  - Tip: read variable declaration from *right-to-left*

const and Pointers

- The syntax with pointers is confusing:

```
int main(int argc, char** argv) {
  int x = 5;                    // int
  const int y = 6;              // (const int)
  y++;

  const int *z = &y;            // pointer to a (const int)
  *z += 1;
  z++;

  int *const w = &x;            // (const pointer) to a (variable int)
  *w += 1;
  w++;

  const int *const v = &x; // (const pointer) to a (const int)
  *v += 1;
  v++;

  return EXIT_SUCCESS;
}
```

constmadness.cc

`const` Parameters

- A `const` parameter *cannot* be mutated inside the function
  - Therefore it does not matter if the argument can be mutated or not
- A non-`const` parameter *may* be mutated inside the function
  - Compiler won't let you pass in const parameters

Make parameters const when you can

```
void foo(const int* y) {
  std::cout << *y << std::endl;
}

void bar(int* y) {
  std::cout << *y << std::endl;
}

int main(int argc, char** argv) {
  const int a = 10;
  int b = 20;

  foo(&a);    // OK
  foo(&b);    // OK
  bar(&a);    // not OK - error
  bar(&b);    // OK

  return EXIT_SUCCESS;
}
```

Poll Everywhere

pollev.com/tqm

- What will happen when we try to compile and run?

A. Output "(2, 4, 0)"

B. Output "(2, 4, 3)"

C. Compiler error about arguments to foo (in main)

D. Compiler error about body of foo

E. We're lost…

poll2.cc

```
void foo(int* const x,
         int* y, int z) {
  *x += 1;
  *y *= 2;
   z -= 3;
}

int main(int argc, char** argv) {
  const int a = 1;
  int b = 2, c = 3;

  foo(&a, &b, c);
  std::cout << "(" << a << ", " << b
   << ", " << c << ")" << std::endl;

  return EXIT_SUCCESS;
}
```

Poll Everywhere

pollev.com/tqm

- What will happen when we try to compile and run?

A. Output "(2, 4, 0)"

B. Output "(2, 4, 3)"

C. Compiler error about arguments to foo (in main)

D. Compiler error about body of foo

E. We're lost…

Can't modify the x, but can modify *x (dereference) poll2.cc

```
void foo(int* const x,
Int ptr → int* y, int z) {
  *x += 1;
  *y *= 2;
   z -= 3;
}

int main(int argc, char** argv) {
  const int a = 1;
  int b = 2, c = 3;

  foo(&a, &b, c);
  std::cout << "(" << a << ", " << b
   << ", " << c << ")" << std::endl;

  return EXIT_SUCCESS;
}
```

Allowed

Copy of int value

Allowed, but change doesn't persist out

Const mismatch