Lecture #2

(aka the Gastrointestinal Lecture)

- Part 1: Basic C++ Concepts
  - Construction & constructors
  - Class Composition
  - Initializer Lists
  - Destruction and destructors
  - Refresher on pointers

- Part 2: On-your-own Study Topics
  - Learning how to use the Visual C++/Xcode debuggers
  - A few final topics you need to know to do Project #1

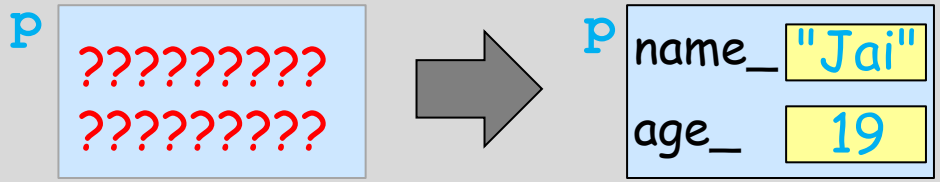
NOTE: I will be teaching class on Friday during your normal discussion sections. Please plan to attend (it won't be a regular discussion section – it'll be CLASS)!

Construction and Destruction

What's the big picture?

Construction is the process of initializing a new object for use when it's first created

```
int main() {
   Person p("Jai", 19);
   ...
}
```



Destruction frees the memory and resources used by an object when its lifetime ends

```
int main() {
   Person p("Jai", 19);
   ...
} // p is destructed
```

Uses:

Construction and destruction are foundational to object oriented programming

Constructors

As we learned in CS31, a constructor is a function that initializes an object when it's first created.

```
class Person {
public:
   Person(string name, int age);
   ...
};
```

```
int main() {
→Person per("Jai", 19);
   ...
}
```

per

name_	"Jai"
age_	19

But a constructor is just piece of the whole construction puzzle!

So, let's learn the details through a set of challenges!

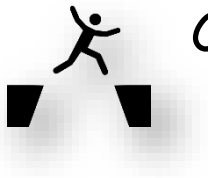
Constructors: W

```
class Constipated {
public:
  Constipated(string name,
              int days) {
    name_ = name;
    days_ = days;
  }

  void hello() {
    cout << "Hi, I'm "
         << name_ << "!\n";
    cout << "I haven't pooped in "
         << days_ << " days!\n";
  }

private:
  string name_;
  int days_;
};
```

```
int main() {
  Constipated d("David",5);
  d.hello();
}
```

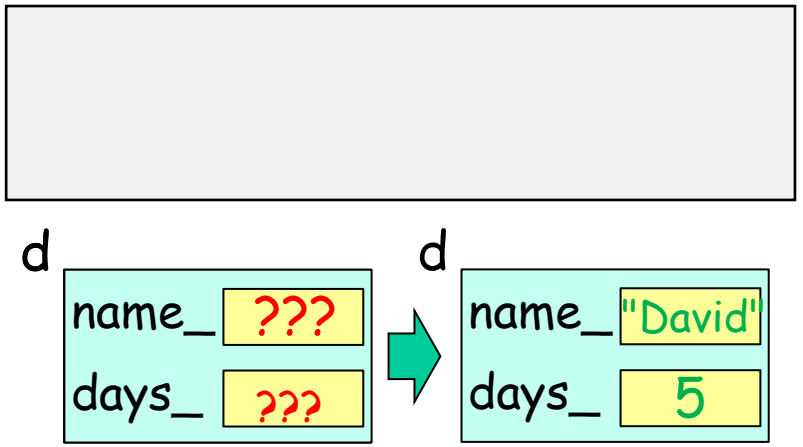
Second, the constructor is given the reserved memory - it initializes the memory to create a valid object.

First, C++ reserves memory to hold our object.



Challenge: What does this program print?

Answer:



Answer:
Hi, I'm David!
I haven't pooped in 5 days!

Constructors: What's Initialized?

```
class Constipated {
public:
  Constipated() {
    cout << "Hi, I'm "
         << name_ << "!\n";
    cout << "I haven't pooped in "
         << days_ << " days!\n";
  }

private:
  string name_;
  int days_;
};
```

```
int main() {
  Constipated d;
  ...
}
```

Interesting! Even though we didn't initialize our name_ member variable before using it...

Challenge: This program prints the following:

name_ was somehow initialized for us before the constructor ran!

But C++ didn't initialize our integer member to zero.

What does this tell us about how/when C++ classes initialize member variables?

```
Answer:
Hi, I'm !
I haven't pooped in 47119 days!
```

Constructors: What's Initialized?

```
class Constipated {
public:
  Constipated() {
    cout << "Hi, I'm "
         << name_ << "!\n";
    cout << "I haven't pooped in "
         << days_ << " days!\n";
  }

private:
  string name_;
  int days_;
};
```

Second, C++ runs the { body } of your constructor.

First, C++ constructs all members that are class types (like string).

C++ won't automatically initialize primitives like ints and doubles.

Ok, so what's going on?

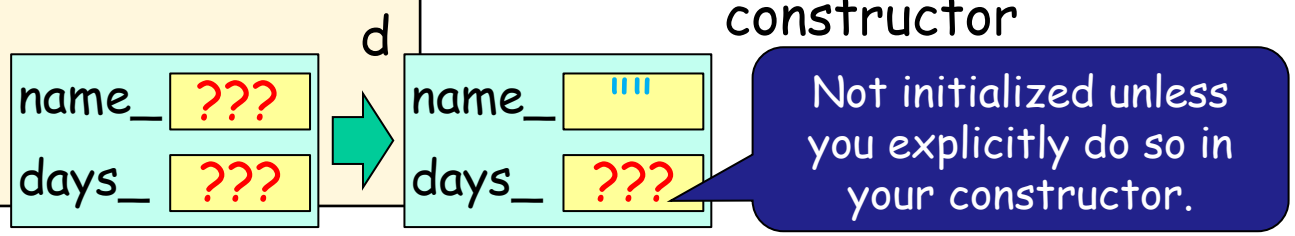
After memory is reserved, construction of an object runs in three phases:

Phase #0:
We'll learn this later ☺

Phase #1:
C++ constructs all non-primitive member variables (e.g., strings) in the order they appear in the class

Phase #2:
C++ runs the { body } of your constructor

```
int main() {
  Constipated d;
  ...
}
```



Constructors:

C++ provides a hidden, empty constructor for you if you don't define your own.

```
class Constipated {
public:
  Constipated() {
  }

  void hello() {
    cout << "Hi, I'm "
         << name_ << "!\n";
    cout << "I haven't pooped in "
         << days_ << " days!\n";
  }

private:
  string name_;
  int days_;
};
```

Ok, but what if you don't define a constructor at all?

If you don't define a constructor, then C++ provides a hidden, empty one for you...

and then initialization goes through the same phases we learned a second ago!

Meme, anyone?

On to the next challenge!

Construction Process

```
class Stomach {
public:
 Stomach()
   { cout << "Empty stomach!\n"; }
};
```

Step #2: This causes C++ to run the Stomach constructor.

```
class Intestine {
public:
 Intestine()
   { cout << "Empty guts!\n"; }
};
```

Step #4: This causes C++ to run the Intestine constructor.

```
class Constipated {
public:
  Constipated()
    { cout << "But can't poop!"; }

private:
  Stomach tummy_;
  Intestine guts_;
};
```

Step #5: Finally, C++ runs the { body } of our Constipated constructor.

Step #1: First C++ initializes our tummy_ member.

Step #3: C++ initializes our guts_ member!

```
int main() {
  Constipated david;
}
```

Challenge: What does this program print?

Answer:

Ok, but why?

Remember our phases?

Phase #1:

C++ constructs all non-primitive member variables (e.g., strings) in the order they appear in the class

Phase #2:

C++ runs the { body } of your constructor.

Answer:
Empty stomach!
Empty guts!
But can't poop!

Process

As you might expect, the construction process works across more than two levels

```
class Banana {
public:
 Banana()
  { cout << "Mmm bplate bananas\n"; }
};
```

#5: So C++ just runs Banana's constructor.

#4: A Banana has no member variables, so there's nothing to initialize first.

```
class Stomach {
public:
 Stomach()
  { cout << "Gurgle gurgle!\n"; }
private:
  Banana banana_;
};
```

#6: Once our banana is constructed, we can run our Stomach constructor.

#3: To initialize a stomach, first C++ initializes its banana_ member variable.

```
class Constipated {
public:
  Constipated(string name) {
    name_ = name;
    cout << name << " is born!\n";
  }

private:
  string name_;
  Stomach tummy_;
};
```

#7: Finally, after all of the members have been constructed, we run our Constipated constructor.

#1: First C++ initializes our name_ member variable.

#2: Then C++ initializes our Stomach variable!

```
int main() {
  Constipated e("Emi");
}
```

Answer:
Mmm bplate bananas
Gurgle gurgle!
Emi is born!

Construction Process

```
class Banana {
public:
 Banana()
  { cout << "Mmm bplate bananas\n"; }
};
```

#3: This is also an example of a default constructor.

```
class Stomach {
public:
 Stomach()
  { cout << "Gurgle gurgle!\n"; }
private:
  Banana banana_;
};
```

#2: It uses their default constructor to do so.

#5: This is an example of class composition.

```
class Constipated {
public:
  Constipated(string name) {
    name_ = name;
    cout << name << " is born!\n";
  }

private:
  string name_;
  Stomach tummy_;
};
```

#4: This is NOT a default constructor, since it requires arguments.

#6: These two member variables are also examples of class composition.

#1: When C++ initializes member variables before the constructor runs...

It's time for a few terms:

class composition

Class composition is when a class holds member variable(s) that are also classes.

default constructor

A default constructor is one which has no required arguments.

Construction Process

```
class Stomach {
public:
 Stomach() { gas_ = 0; }
 void fart() {
   while (gas_-- > 0) cout << "pfft";
 }
 ...
};
```

#4: So it has a valid state (e.g., gas_ is set to zero).

```
class Gassy {
public:
  Gassy(string name) {
    name_ = name;
    belly_.fart();
  }

private:
  Stomach belly_;
  ...
};
```

#2: Before running the constructor { body }?

#5: Otherwise we can't safely use it in our constructor!!

#3: We have to construct our belly_ member variable first...

#1: Why does C++ construct member variable(s)...

You might wonder: Why does C++ construct member variables before running the constructor?

C++ does this to ensure the member variables are initialized before they're used by the code in the constructor { body }.

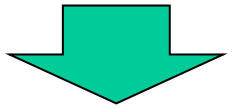
Construction Process

```
class Stomach {
public:
 Stomach(int oz_of_gas)
  { gas_ = oz_of_gas; }
 ...
};
```

#1: To construct a Stomach, we must pass in a parameter to its constructor.

```
int main() {
  Stomach s;  // ERROR!
  ...
}
```

#2: But we currently don't specify an argument when constructing variable s.



```
int main() {
  Stomach s(100);  // OK!
  ...
}
```

If we add a parameter to our constructor, we're all set!

Challenge: This program doesn't compile – what's the problem with it?

Answer:

Ok – file that fact away.

Answer: Our Stomach's constructor requires a parameter... but our main function doesn't pass a parameter!

Co[illegible]Process

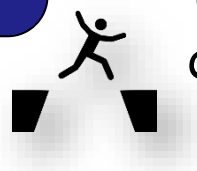
```
class Stomach {
public:
 Stomach(int oz_of_gas)
  { gas_ = oz_of_gas; }
 ...
};
```

Again, to construct a Stomach, we must pass in a parameter to its constructor.

```
class Gassy {
public:
  Gassy(string name)
  {
    name_ = name;
  }

private:
  Stomach belly_;
  string  name_;
};
```

But our Gassy class doesn't have any way to specify an argument for our Stomach!

```
int main() {
  Gassy d("David");
}
```



Challenge: This program doesn't compile – what's the problem with it?

(Hint: Think about our last slide!)

Answer:

How do we fix this?

Answer: Our Stomach's constructor requires a parameter... but our Gassy class doesn't pass any parameter!

Construction: Initializer Lists

```
class Stomach {
public:
 Stomach(int oz_of_gas)
  { gas_ = oz_of_gas; }
 ...
};
```

#2: when they require argument(s) for construction.

#5: And once we do that, we can run Gassy's constructor { body }.

#3: This means: Every time we construct a Gassy object...

```
class Gassy {
public:
  Gassy(string name)
    : belly_(5)
  {
    name_ = name;
  }

private:
  Stomach belly_;
  string  name_;
};
```

#1: This is an initializer list - it is used to initialize member variable(s)...

#4: we must first construct its belly_ member variable... by passing in a value of 5 to its constructor.

```
int main() {
  Gassy d("David");
}
```

To fix this, our Gassy class needs to pass a value into belly_'s constructor.

e.g., let's say we want every Gassy person to start with 5 oz of gas in their belly.

Here's the syntax!

It's time for a new term:

initializer list

Code that explicitly passes in value(s) to the constructor(s) of one or more member variables

Construction: Initializer Lists

#3: This runs our Stomach constructor.

```
class Stomach {
public:
  Stomach(int oz_of_gas)
    { gas_ = oz_of_gas; }
  ...
};
```

#5: This runs the string class's constructor.

```
class string {
public:
  string() {
    // code to initialize
    // our string to ""
  }
  ...
};
```

```
class Gassy {
public:
  Gassy(string name)
    : belly_(5)
  {
    name_ = name;
  }

private:
  Stomach belly_;
  string  name_;
};
```

#6: Finally, we can run the { body } of the Gassy constructor.

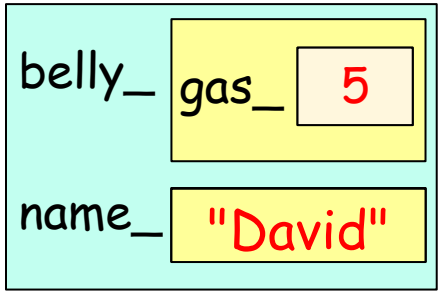
#1: As we learned, C++ first constructs all member variables that are class types.

#2: So first we'll construct our belly_ member.

#4: We're not done yet, we also must construct our name_ member variable.

```
int main() {
  Gassy d("David");
}
```

d



tializer Lists

```
class Stomach {
public:
 Stomach(int oz_of_gas)
    : gas_(oz_of_gas)
 { /* empty now */ }
private:
  int gas_;
};
```

```
class Gassy {
public:
  Gassy(string name)
    : belly_(5), name_(name)
  {
    name_ = name;
  }
private:
  Stomach belly_;
  string  name_;
};
```

```
int main() {
  Gassy d("David");
}
```

For instance, if we like we can initialize our name_ member here!

1st

2nd

Now we can get rid of this code here.

1st

2nd

We may use our initializer list to initialize any number of member variables!

Just add them one after the other, separated by commas.

A best practice is to order the variables in the initializer list the same order as the variables are defined in the class.

Finally, initializer lists can be used to initialize primitives (e.g., ints, doubles) too!

Overloaded Constructors

```
class Constipated {
public:
  Constipated(string name) {
    name_ = name;
    days_ = 0;
  }

  Constipated(string name,
              int days) {
    name_ = name;
    days_ = days;
  }

private:
  string name_;
  int days_;
};
```

```
int main() {
  Constipated d("Zhenda");
  Constipated c("Carey", 5);
}
```

These are treated as two entirely different functions!

This is an example of function overloading (in this case, constructor overloading).

C++ figures out which version to call based on the # and types of arguments.

You might not have expected it, but...

a class can have as many constructors as you like!

Each constructor must have a different number/types of arguments.

It's time for a new term:

function overloading

Function overloading is when we define multiple functions with the same name but different numbers/types of parameters

When Are Constructors Called?

```
int main() {
  Constipated  carey;

  Constipated  arr[52];

  Constipated  *ptr;

  ptr = new Constipated;

  for (int i=0;i<5;++i) {
    Constipated temp;
      ...
  }
  ...
}
```

A constructor is called any time you create a new object.

A constructor is called N times (once for each element) for an array of N objects.

The constructor is *not called* when you just define a pointer variable!

A constructor *is* called when you use new to dynamically create a new object.

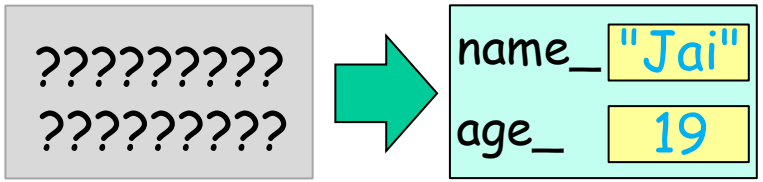
If a variable is declared in a loop, a new version is constructed during EVERY iteration!

Challenge: Identify all the places constructors are called in this code.

Hint: Constructors are ONLY called to initialize new objects.

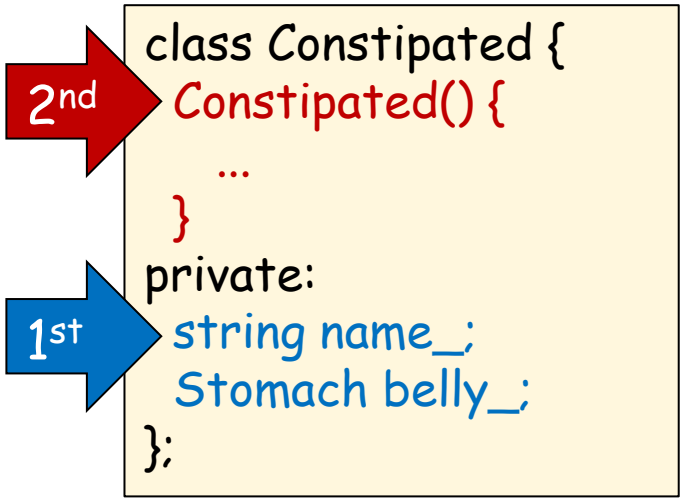
Constructor Summary, Part 1

A constructor is a function that initializes an object when it's first created.



First, C++ calls the constructors for an object's non-primitive member variable(s) in the order they're defined

Then C++ runs the constructor's { body } to complete initialization



When we have multiple levels of class composition (e.g., Gassy holds a Stomach which holds a Banana), construction follows the same pattern

Constructor Summary, Part 2

We use an initializer list to initialize member variables that require parameters for construction

```
class Stomach {
public:
  Stomach(int gas) { ... }
private:
  int gas_;
};
```

```
class G
public:
  Gassy() : belly_(5) { ... }
private:
  Stomach belly_;
};
```

initializer list

Constructors run every time we define a new object

But DO NOT run if we just define a pointer (since there's no object!)

```
  Gassy c("Carey");

  Gassy *ptr;
  ptr = new Gassy("David");
};
```

constructor runs here

but not here!

and here

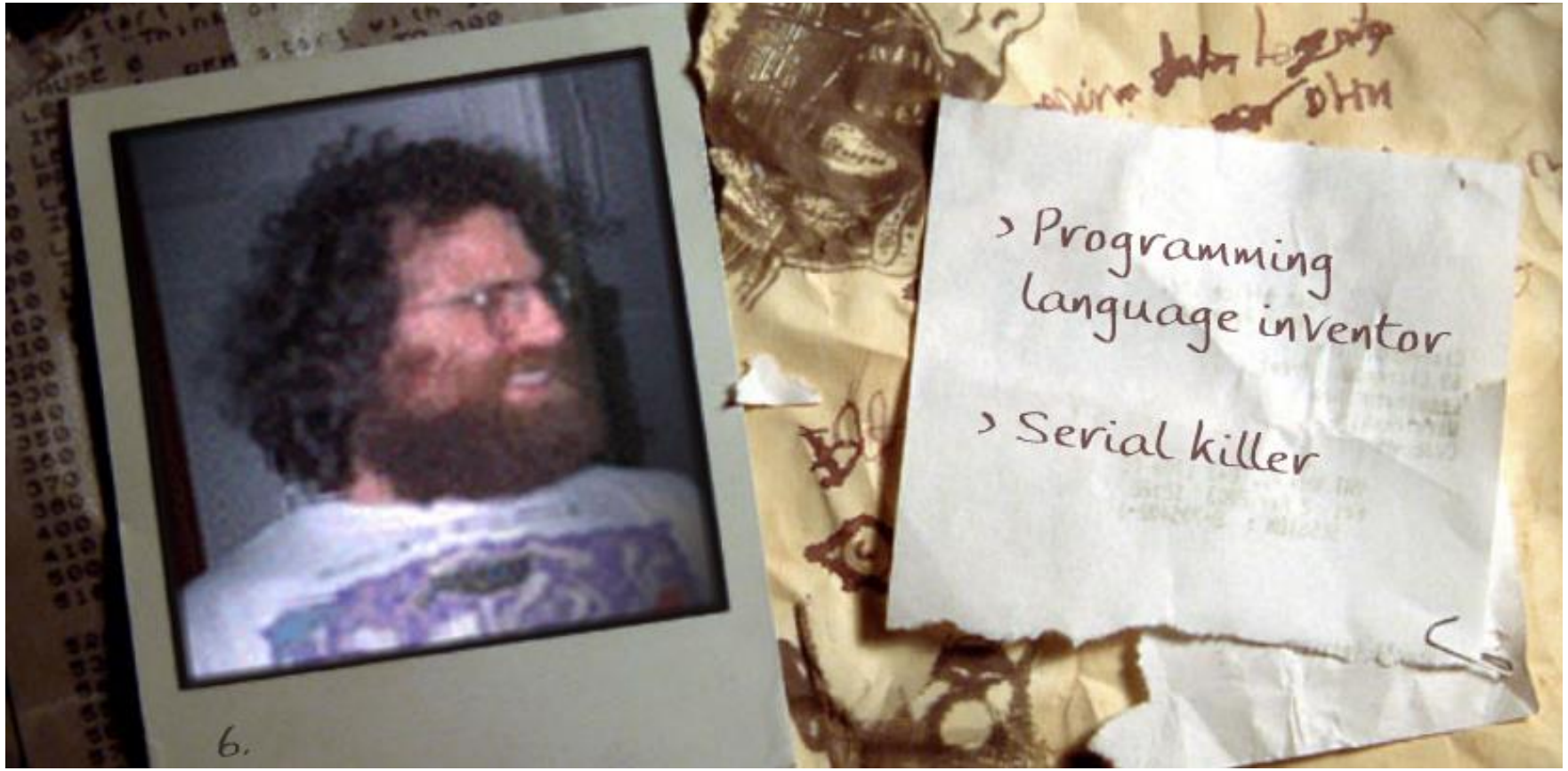
Classes can have more than one constructor (w/diff parameters)

This is called constructor overloading

```
class Gassy {
public:
  Gassy() { ... }
  Gassy(int start_gas) { ... }
  Gassy(string name, int start_gas) { ... }
};
```

Can you guess?

Is this guy a programming language inventor or a serial killer?

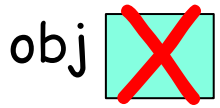


Philip Wadler. He invented the “Haskell” language.

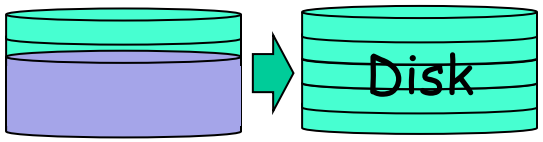
Destructors

A destructor function frees all the resources that an object allocates during its lifetime.

An object often reserves system resources (e.g., memory, disk) as it runs.



Destructors ensure those resources are freed up when an object goes away.



Let's learn the details through a set of challenges!

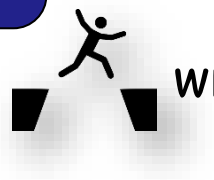
Destructors

```
class Song {
public:
 Song() {
   tmp_file_ = create_temp_file();
 }

 void play_song(string song_url) {
    download(song_url, tmp_file_);
    play(tmp_file_);
 }

 void done() { remove(tmp_file_); }

 ...

};
```

#1: Our Song object allocates a temporary file on disk.

```
int main() {
  Song s;
  s.play_song("spotify.com/song/317");

  // temp file never deleted!
} // s's destructor runs and deletes temp file!
```

#2: But our programmer forgot to call s.done() to delete it, so it's never deleted!



Challenge: What's wrong with this code?

Answer:

Destructors ensure that resources allocated by an object are released when the object's lifetime ends.

Answer: It never deletes the temporary file it creates.

Destructors

```
class Song {
public:
 Song() {
   tmp_file_ = create_temp_file();
 }

 void play_song(string song_url) {
    download(song_url, tmp_file_);
    play(tmp_file_);
 }

 void done() { remove(tmp_file_); }

 ~Song() { remove(tmp_file_); }

};
```

```
int main() {
  Song s;
  s.play_song("spotify.com/song/317");

  // temp file never deleted!
} // s's destructor runs and deletes temp file!
```

Destructors ensure that resources allocated by an object are released when the object's lifetime ends.

#1: But if we add a destructor, we guarantee when the object goes away, its resources go away too.

#2: Now our temporary file will automatically be deleted!

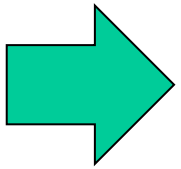
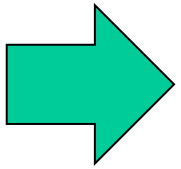
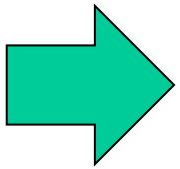
Destructors: Another Example

```
class Joke { ... };

class Comedian {
public:
 Comedian() { joke_ = nullptr; }

 void brainstorm() {
   joke_ = new Joke("USC education");
 }

 ~Comedian() { delete joke_; }

private:
  Joke *joke_;
};
```

In CS32, we'll primarily be using destructors to free dynamically allocated memory.

But there are other uses too - let's learn all of them.

When Must You Have a Destructor?

Any time a class allocates a system resource…		Your class must have a destructor that…
Reserves memory using the *new* command		Frees the allocated memory with the *delete* command
Opens/Creates a disk file		Closes/Deletes the disk file
Connects to another computer		Disconnects from the other computer

Destructors

```
class Stomach {
public:
 ~Stomach()
   { cout << "Bye stomach!\n"; }
};
```

```
class Intestine {
public:
 ~Intestine()
   { cout << "Bye guts!\n"; }
};
```

```
class Constipated {
public:
  ~Constipated()
    { cout << "Still no poop!"; }

private:
  Stomach tummy_;
  Intestine guts_;
};
```

```
int main() {
  Constipated d;
}
```

Challenge: What does this program print?

Answer:

```
Still no poop!
Bye guts!
Bye stomach!
```

But why?

Destructors

#4: The destruction of tummy_ runs this destructor (after running guts_ destructor).

```
class Stomach {
public:
  ~Stomach()
   { cout << "Bye stomach!\n"; }
};
```

#3: The destruction of guts_ runs this destructor (after running Constipated's destructor).

```
class Intestine {
public:
  ~Intestine()
   { cout << "Bye guts!\n"; }
};
```

#1: First, the object's destructor { body } runs to completion.

```
class Constipated {
public:
  ~Constipated()
    { cout << "Still no poop!"; }

private:
  Stomach tummy_;
  Intestine guts_;
};
```

#2: Then, starting from the bottom, we destruct each member variable.

```
int main() {
  Constipated d;
}
```

Destruction of an object d runs in three phases:

Phase #0:
C++ runs object d's destructor { body } first

Phase #1:
C++ destructs all non-primitive member variables in the reverse order they appear in the class

Phase #2:
We'll learn this later ☺

After of the destructors have run, the object's RAM is freed.

Destructors

```
class Stomach {
public:
 ~Stomach()
   { cout << "Bye stomach!\n"; }
};
```

```
class Intestine {
public:
 ~Intestine()
   { cout << "Bye guts!\n"; }
};
```

```
class Constipated {
public:
   ~Constipated() {
   }

private:
   Stomach tummy_;
   Intestine guts_;
};
```

If you don't define a destructor...

then C++ provides a hidden, empty one for you...

and it goes through the same destruction phases we learned a second ago!

And tummy_ will have its destructor run second.

So guts_ will have its destructor run first.

Destructors

As you might expect, the destruction process works across more than two levels

```
class Banana {
public:
  ~Banana()
    { cout << "Mmm rotten bananas\n"; }
};
```

#6: This runs the destructor { body } for our Banana object.

#7: Since Banana has no member variables, there's nothing more to do here.

```
class Stomach {
public:
  ~Stomach()
    { cout << "Gurgle gurgle!\n"; }
private:
  Banana banana_;
};
```

#4: This runs the destructor { body } for our Stomach object.

#8: Since Stomach has no more member variables, there's nothing more to do here.

#5: Then C++ destructs Stomach's members from bottom to top!

```
class Constipated {
public:
  ~Constipated() {
     cout << "Finally relived!\n";
   }

private:
  Stomach tummy_;
};
```

#2: First, C++ runs the destructor { body } for our object d.

#3: Then C++ fully destructs each of d's members, one at a time, from bottom to top.

#9: Since Constipated has no more member variables, there's nothing more to do here.

#1: d is destructed here, when its lifetime ends.

```
Finally relieved!
Gurgle gurgle!
Mmm rotten bananas
```

```
int main() {
  Constipated d("David");
}
```

Destructors

```
class Stomach {
public:
 Stomach() { gas_ = 0; }
 void eat() { gas_++; }
 ~Stomach() { ... }
};
```

```
class Constipated {
public:
  ...
  ~Constipated() {
    tummy_.eat();
    cout << "Ate one last meal!";
  }

private:
  Stomach tummy_;
};
```

#1: The destructor is using/changing the object's member variables!

#3: After the destructor finishes, it's safe to destruct our member variable, since they will no longer be referenced.

#2: So these variables have to be valid until the outer destructor finishes running!

Challenge: Why does C++ destruct a class's member variables *after* the class's destructor runs?

Here's a hint!

Answer:

Once the outer destructor has finished, then the members can be safely destructed.

Answer: A class's destructor is allowed to use the class's member variables!

Construction vs Destruction Order

```
class Banana {
public:
  Banana() { ... }
  ~Banana() { ... }
};
```

c'tor { ... } run first

d'tor { ... } run 3rd

```
class Stomach {
public:
  Stomach() { ... }
  ~Stomach() { ... }
private:
  Banana banana_;
};
```

c'tor { ... } run 2nd

d'tor { ... } run 2nd

```
class Gassy {
public:
  Gassy() { ... }
  ~Gassy() { ... }
private:
  Stomach tummy_;
};
```

c'tor { ... } run 3rd

d'tor { ... } run first

Let's contrast construction order vs destruction order.

```
int main() {
  Gassy g;
  ...
} // g is destructed
```

A good way to remember this is construction runs top-to-bottom, and destruction runs bottom-to-top.

Destructors

```
void dingleberry() {
  Gassy a;

  for (int j=0;j<10;j++){
    Gassy b;
    ...
  }

  Gassy *c = new Gassy;
  Gassy *d = new Gassy;
  Gassy *e;

  delete c;

  Gassy f[20];

}
```

#1 (at the `}` of the for loop)

#2 (at `delete c;`)

#3 (at the `}` of dingleberry)

#4 (at the `}` of dingleberry)

Since there's no delete for variable d, it's destructor is NEVER called!

Only objects can be destructed, and a pointer is NOT an object – so no destructor runs for it.

All 20 objects in the f array have their destructors called here

Challenge: Identify all the places destructors are called in this code.

Here are the two rules:

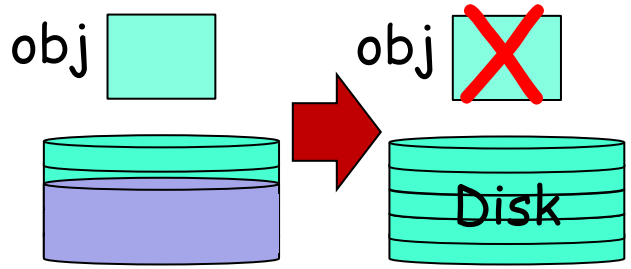
Local variables are destructed when their lifetime ends

Dynamically-allocated objects are destructed only if/when delete is used on them

Answers:
#1: b's destructor is called here (during each loop iteration!)
#2: the object pointed to by c is destructed here
#3: f's 20 destructors are called here
#4: a's destructor is called here

Destructor Summary, Part 1

A destructor function frees all the resources that an object allocates during its lifetime



First, C++ runs the object's destructor { body } to free all resources held by the object

Then C++ goes through non-primitive member variable(s) in reverse order and destructs each of them

```
class Constipated {
  Constipated() {
    ...
  }
private:
  string name_;
  Stomach belly_;
};
```

1st

2nd

When we have multiple levels of class composition (e.g., Gassy holds a Stomach which holds a Banana), destruction follows the same pattern

Destructor Summary, Part 2

Destructors run every time an object's lifetime ends, e.g.

when we exit a { block } where a local variable was defined

and when we delete an object through a pointer

```
int main() {
  Gassy local_var("Carey");

  Gassy *g1 = new Gassy("Misha");
  Gassy *g2 = new Gassy("Len");

  delete g1;  // Misha's d'tor runs

} // local_var's destructor runs
```

But if you forget to delete an object, it'll never be destructed!

Pointers

Are you saying you are a CS mayor and you are still confused by how C++ pointers work?

Addresses and Pointers…

What's the big picture?

Addresses and pointers are one of the most confusing topics for new CS students.

Rather than explain them here, let's learn them via a series of challenges.

Uses:

Pointers enable dynamic memory allocation, complex data structures, and argument passing.

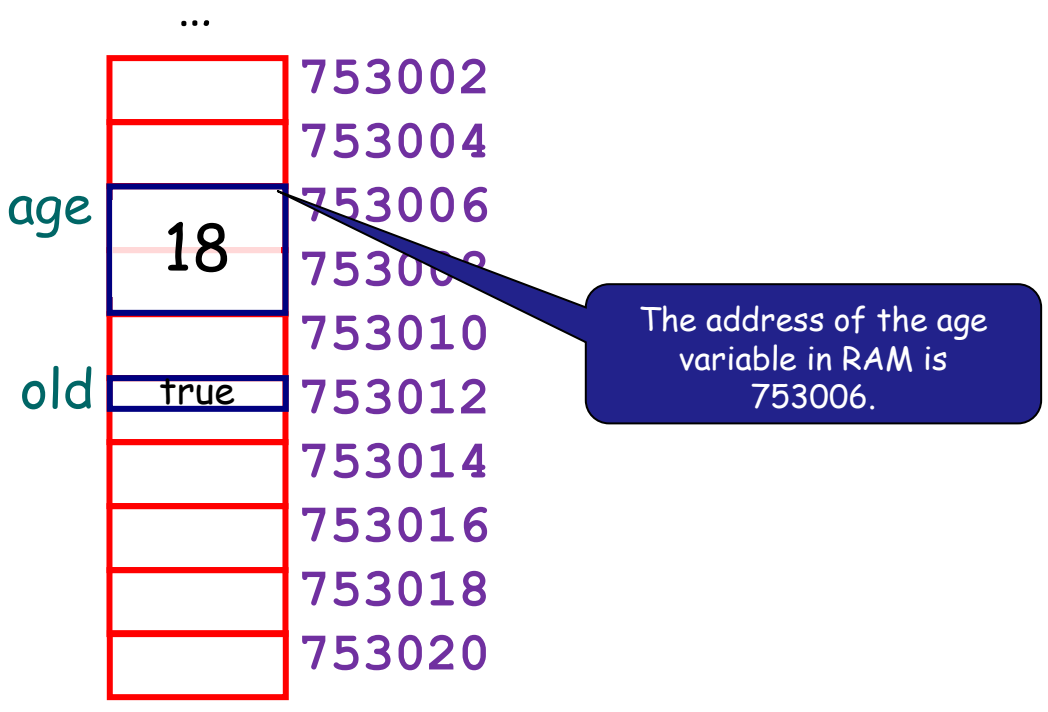
Addresses

```
int main() {
  int age = 18;
  bool old = true;

  cout << &age; // prints 753006
  cout << &old; // prints 753012
}
```

The & operator tells us where a variable is located in RAM - its address.

This code prints the addresses of the age and old variables

1. What is an address?



2. How do we get a variable's address?

3. Is an address a variable?

Answers:
#1: An address is a number identifying the starting location of a variable in RAM
#2: We can obtain an address for a variable using the & operator.
#3: No! An address is the location of a variable - it takes up no storage, it's just a location!

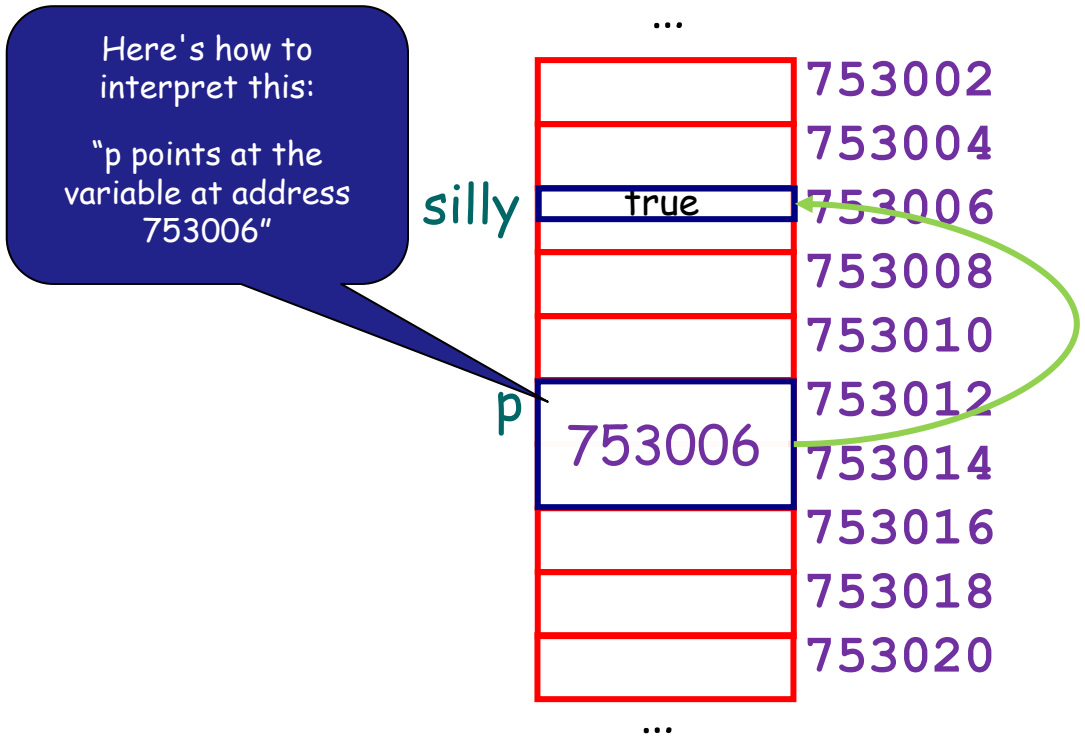
Pointers

```
int main() {
  bool silly = true;

  bool *p;
  p = &silly;
}
```

Consider this code which defines a pointer called p and assigns its value

1. Is a pointer like p a variable?

2. What types of values can pointers hold?



Answers:
#1: Yes, pointers are variables - they hold values just like other variables
#2: You can only store addresses of variables inside of pointer variables

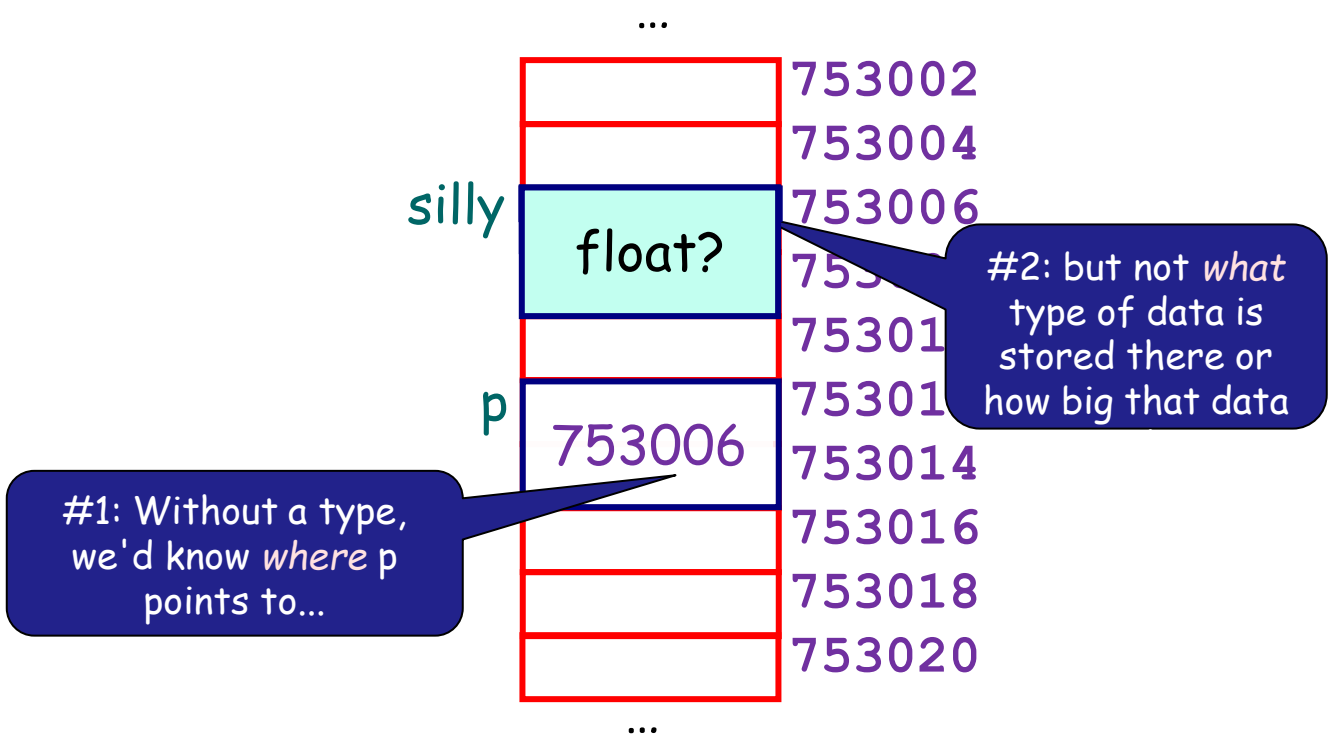
Pointers

This type tells us that p points at a Boolean variable.

```
int main() {
  bool silly = true;

  bool *p;
  p = &silly;
}
```

We read the type from right to left: "Variable p is a pointer to a bool value."

Do pointers have types? Why or why not?



To understand the type of your pointer variable, simply read your declaration from right to left...

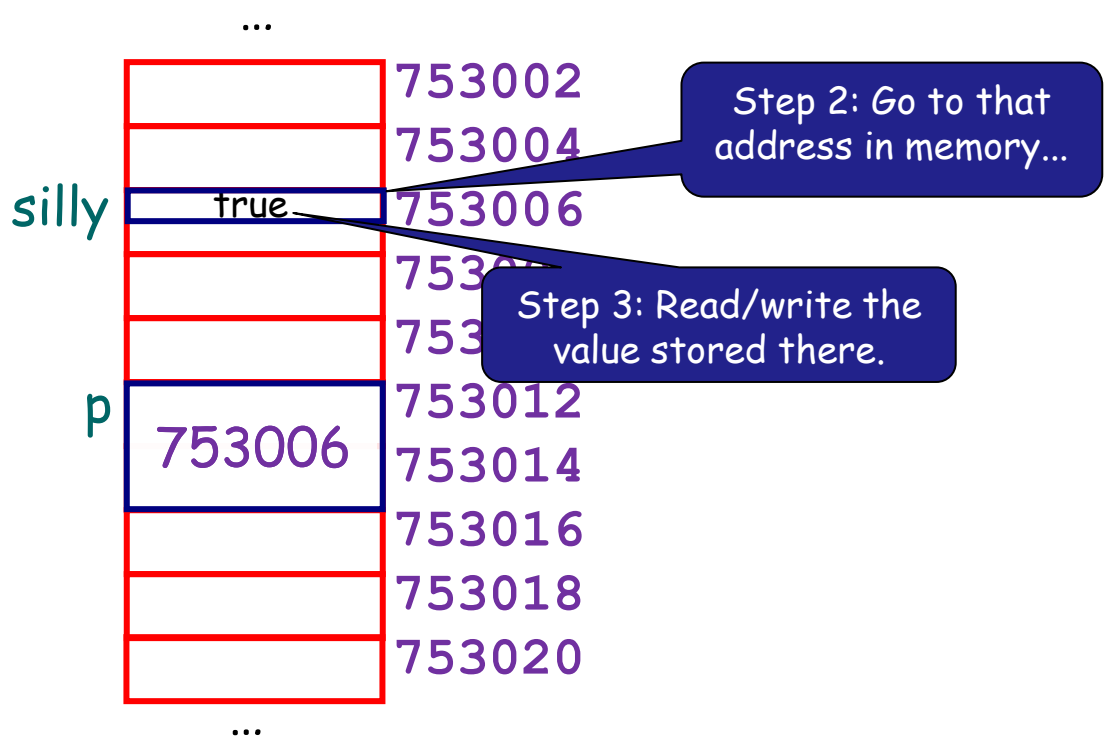
Answer: Yes, every pointer variable has a type!
A pointer variable's type tells us what type of variable it "points" at.

Dereferencing Pointers

```
int main() {
  bool silly = true;

  bool *p;
  p = &silly;

  cout << *p;
}
```

Step 1: Fetch the address stored in the pointer variable p…

The * operator takes the _____ stored in a pointer variable and uses it to read/write the ____ stored at that location in memory.



Step 2: Go to that address in memory…

Step 3: Read/write the value stored there.

cout << *p;

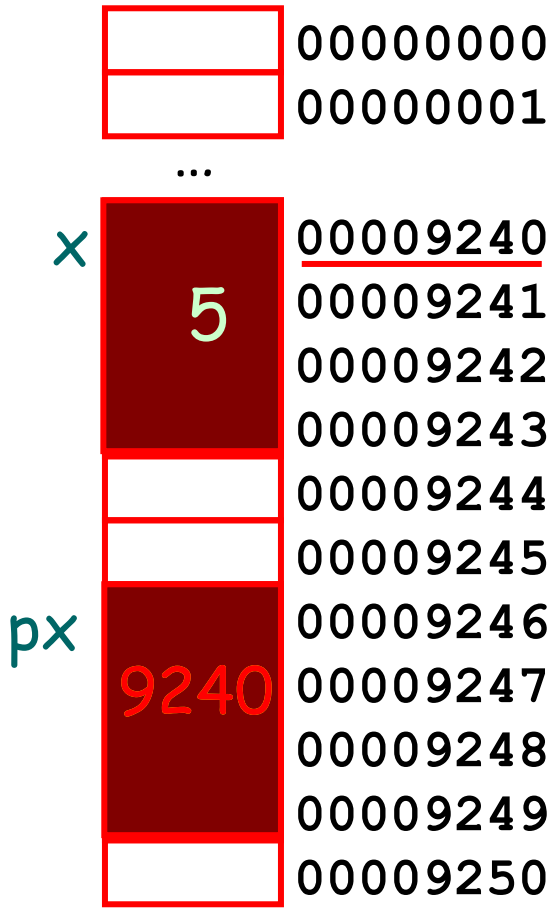
cout << *753006;

cout << true;

Dereferencing Pointers

9240

```
void set(int *px) {
 *px    = 5;
}

int main() {
 int x = 1;
    9240
 set(&x);

 cout << x;    // prints 5
}
```

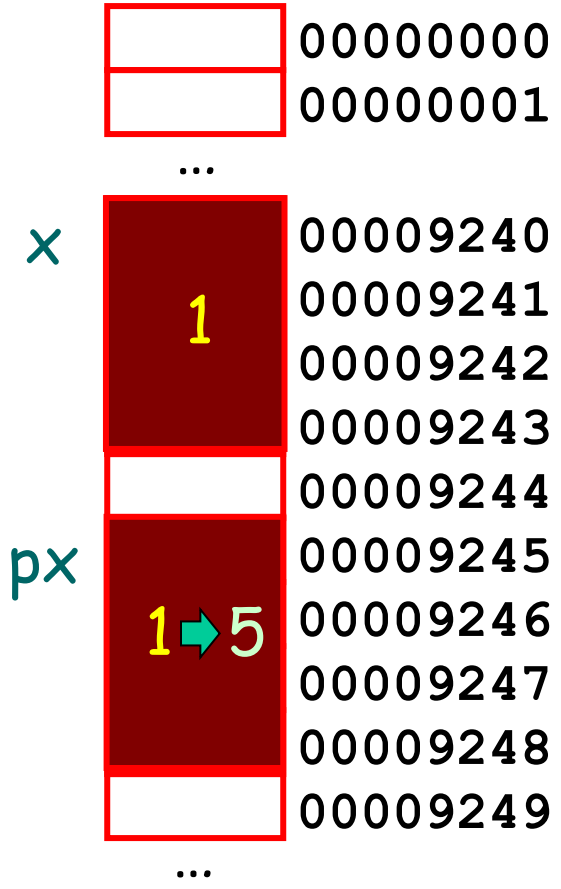


A common use case for pointers is enabling a function to modify a variable defined in another function.

What if We Didn't Use Pointers?

```
                    1
void set(int   px) {
  px     = 5;
}

int main() {
 int x = 1;
       1
 set( x);

 cout << x; // prints 1
}
```

What does it print if we don't use pointers?



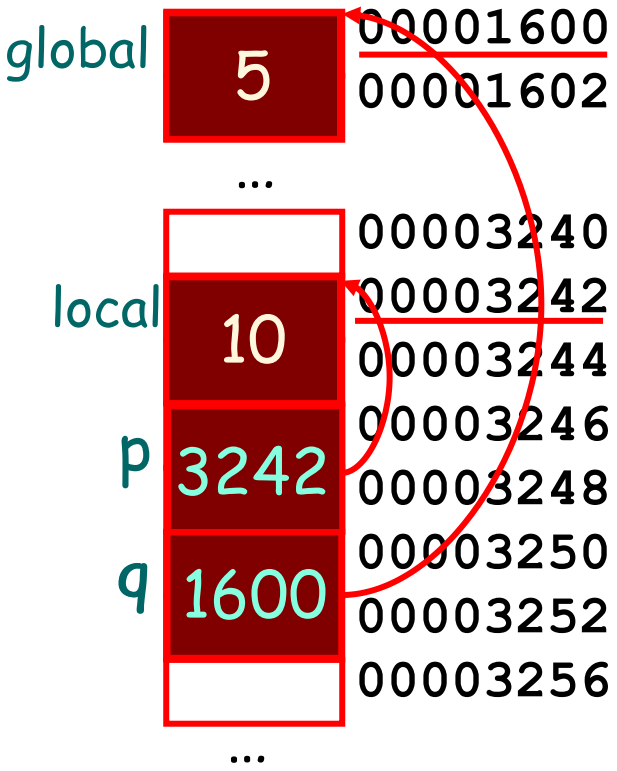
Answer: The assignment in the set() function modifies the local px variable, but does not change the original variable x.

A Common Pointer Mistake!

#1: This changes the local pointer variable q

#2: But does nothing to change p, a separate pointer variable.

```
int global = 5;

void foo(int *q) {
   q = &global;
}

int main() {
   int local = 10;
   int *p = &local;

   foo(p);
   cout << *p;
}          // prints 10
```



What does this program print? Be careful!

Moral: Assigning a pointer in a function (e.g., q = &global;) will only change the local pointer!

Answer: 10

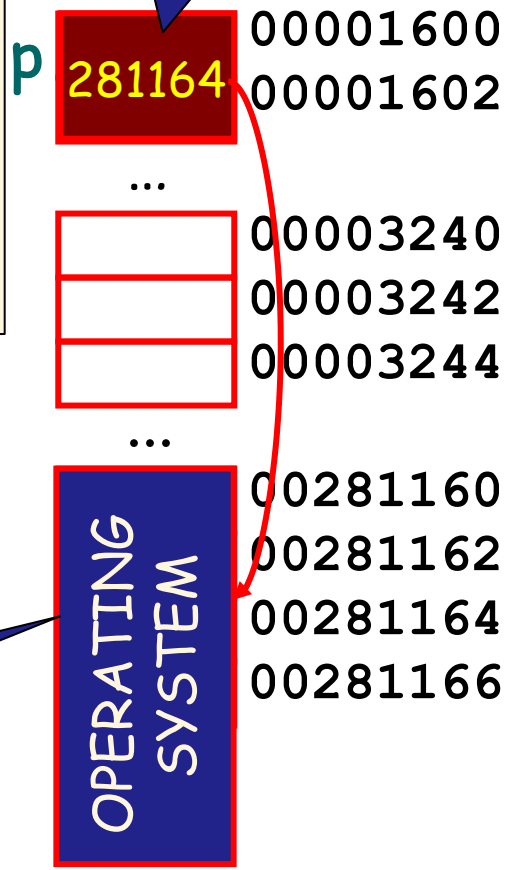
Another C[illegible]ter Mistake!

```
int main() {
  int *p;

  *p = 42;
  ...
```

Uninitialized pointers hold a random value!

Instead, initialize p to null ptr right away:

int *p = nullptr;



So it could point anywhere in memory!

What's wrong with this program?

Pro tip: Always initialize pointers to nullptr immediately when you define them!

Why? If you use * on a null pointer, your program will crash immediately, and you'll find the bug ASAP!

Answer: If you don't initialize a pointer variable then it holds a RANDOM address! So this program randomly corrupts memory!

Pointers vs References

When you pass a variable by reference to a function, what really happens?

```
void set(int &val) // val is a ref
{
   val = 5;
}

int main()
{
   int x = 1;
   set(x);
   cout << x;
}
```

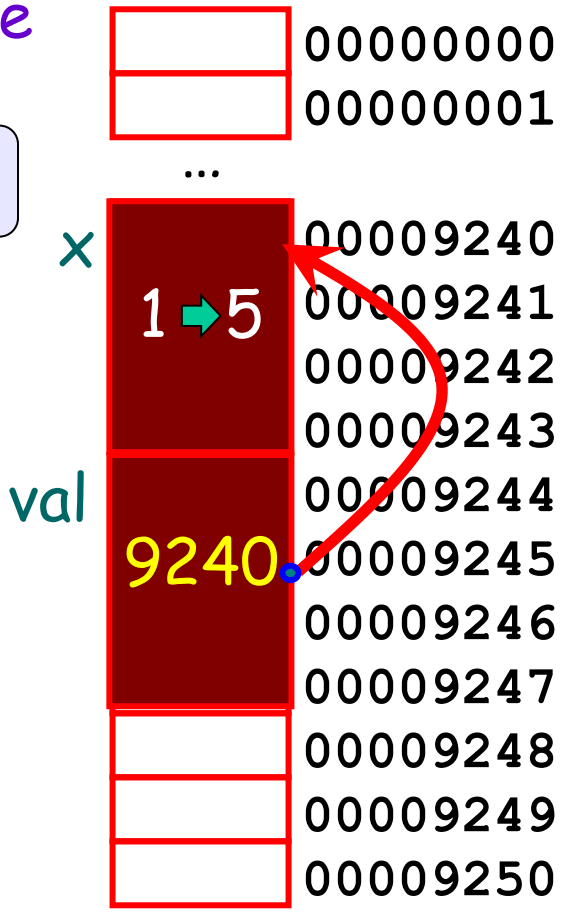
#1: Since the set function accepts a reference…

#4: Since val points to our original variable, x, this line actually changes x!

#2: It looks like we're just passing the value of x, but in fact…

9240

#3: This line is really passing the address of variable x to set…



In fact, a reference is just a simpler notation for passing by a pointer!

(Under the hood, C++ uses a pointer)

A Few Final Topics

Let's talk about four final topics that you'll need in order to solve your Project #1…

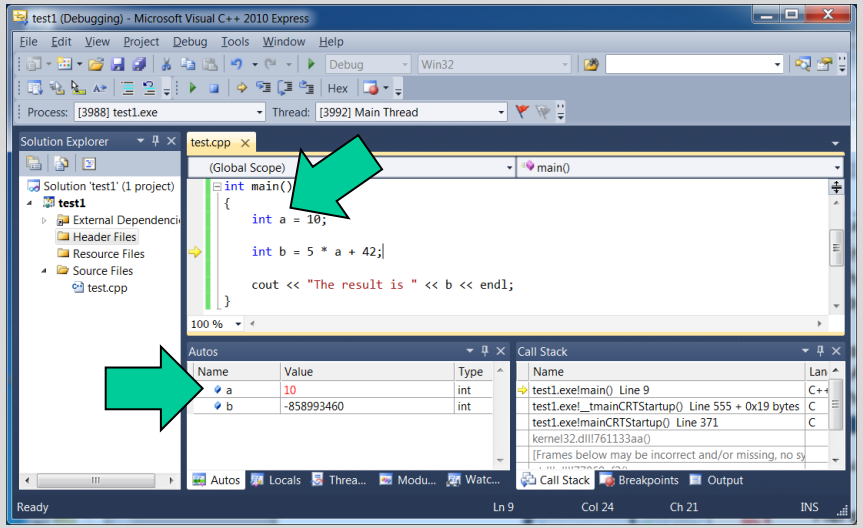
Debugging

Learning to Use the C++ Debugger

What's the big picture?

Visual Studio/Xcode debuggers help you find bugs 10x faster!

You can trace programs line-by-line and see the value of every variable (like we do in slides)!



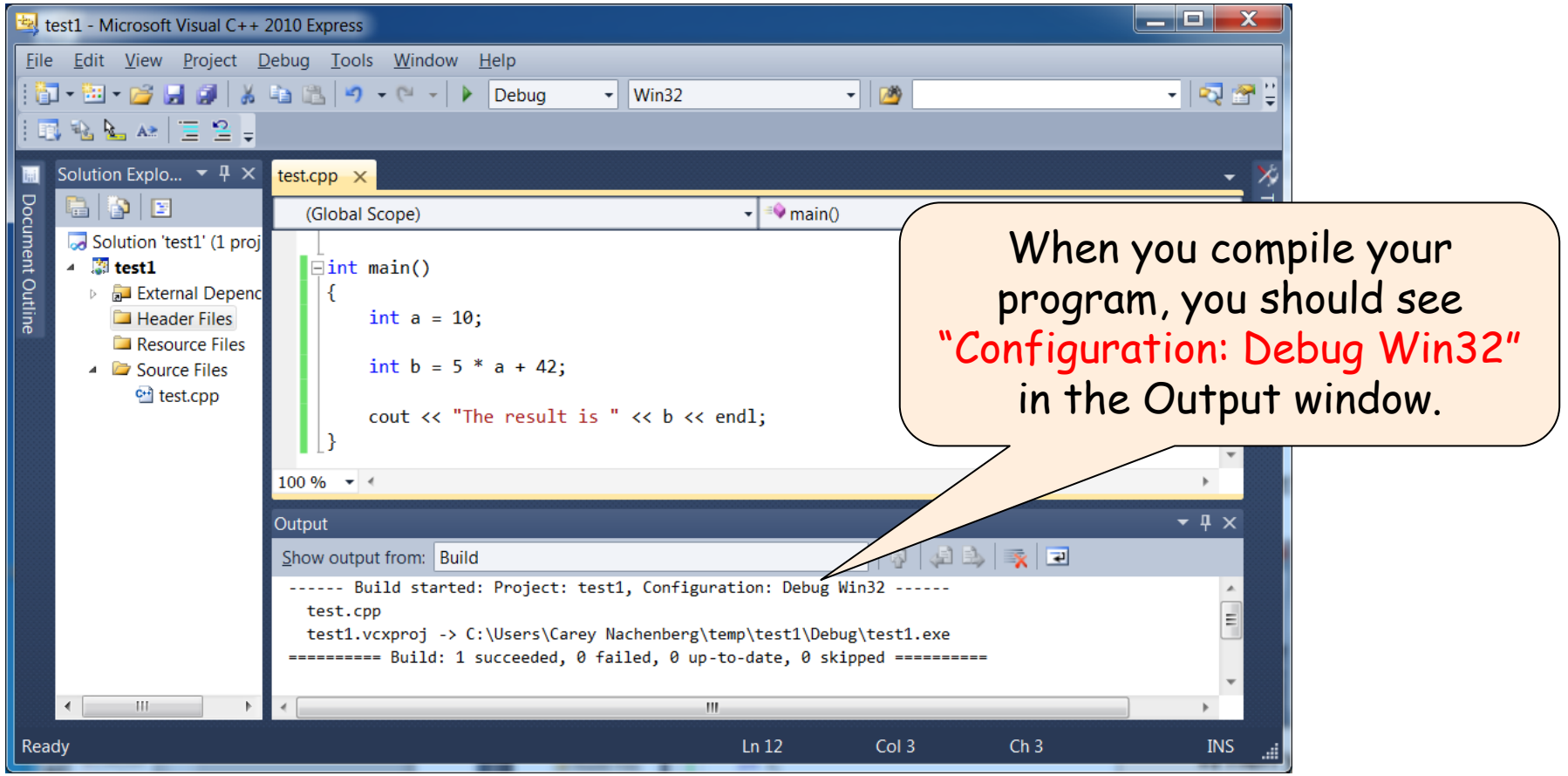
It takes about 10 mins to learn how the Visual Studio/Xcode debuggers work, so get going!

Uses:

Use the debugger to quickly figure out why your code is crashing, hanging or giving the wrong result.

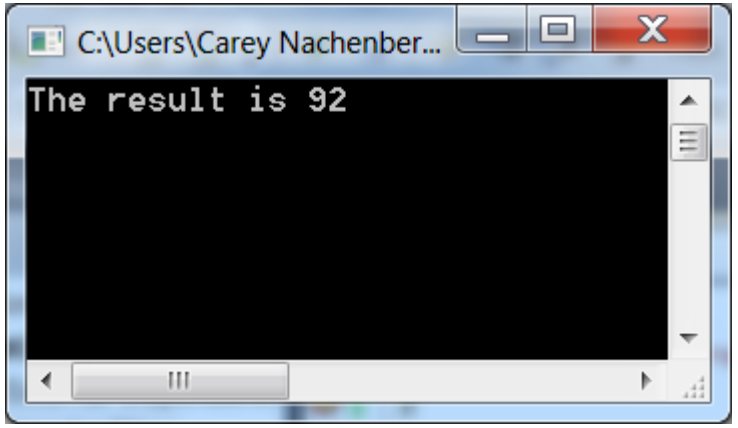
Debugging

By default, Visual Studio compiles all programs in "debug mode" so you can use the debugger with them right away.

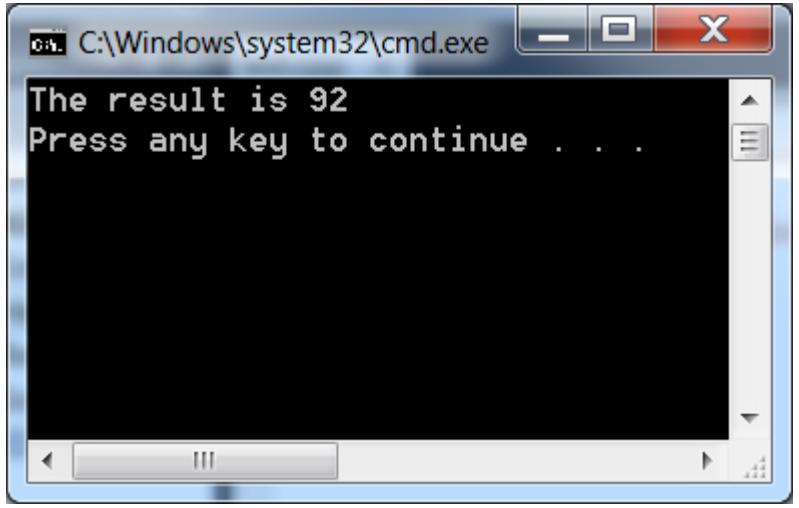


Debugging

You can run your program from start to finish by hitting either F5 or Ctrl-F5.



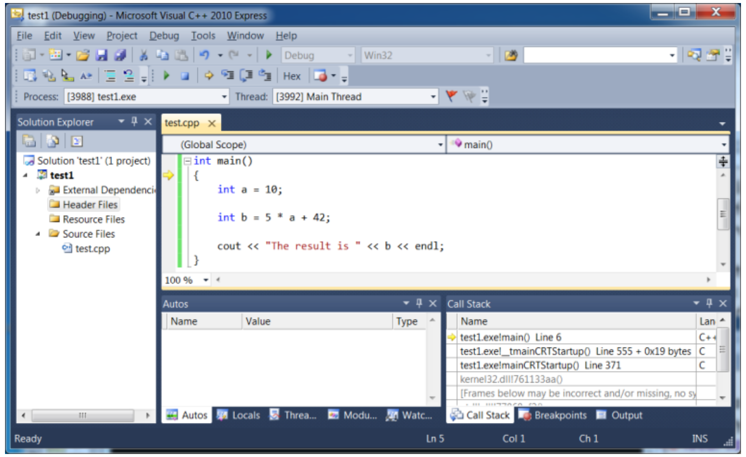
If you run your program by hitting F5, Visual Studio will immediately close the debug window after your program finishes.

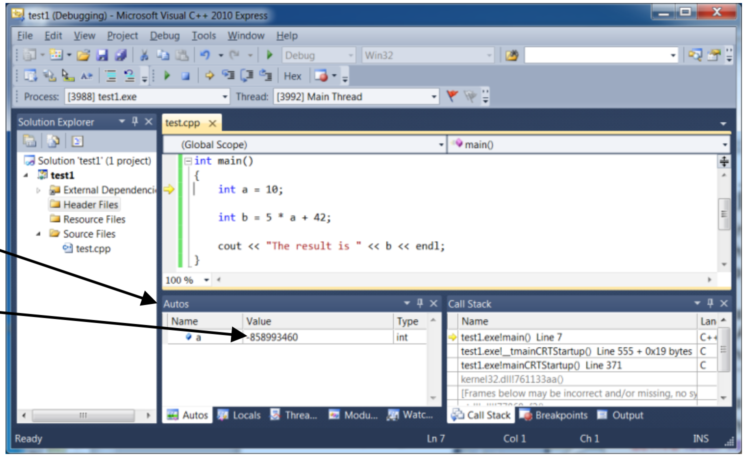


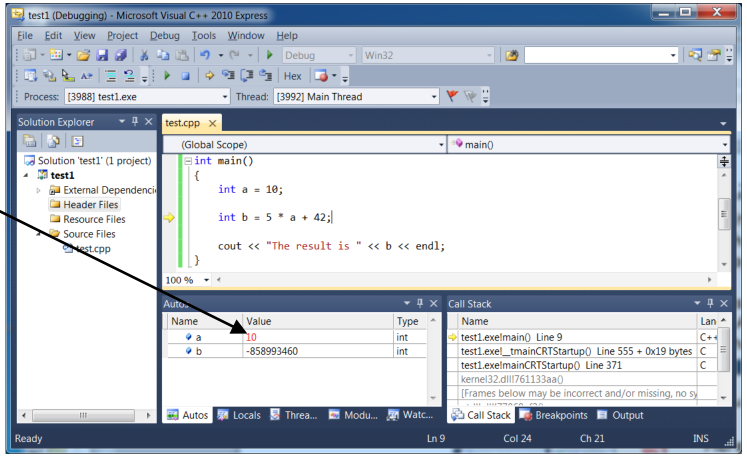
If you run by hitting Ctrl-F5, Visual Studio will print "Press any key to continue..." and leave the debug window on the screen when your program completes.

Debugging

- To start your program and debug one line at a time, hit F10 after you finish compiling it.
- You can then continue stepping through your program one statement at a time using the F10 and F11 keys.
- When you first start debugging your program by hitting F10, you'll see the little yellow arrow next to the first line of your program (see upper-right corner). The arrow points to the line of code that's ABOUT to execute (that line hasn't been run yet).
- Hitting F10 again takes you to the first line inside the function (see middle-right)
- Notice that Visual Studio has a sub-window where it displays the values of your active local variables – these are called "auto" variables.
- Since we haven't run the current line of code yet (a = 10;) to initialize a value, a is currently random.
- If we trace once more by hitting F10, we'll run the a=10; statement and see that a's value is now 10 (hi-lited in red because the value changed)
- Also notice that variable b is now active, and has a random value too since the line b = 5 * a + 42; hasn't run yet.
- If you were to hit F10 again, then b's value would change to 92 and be shown in red. a's value would stay 10, but no longer be highlighted.
- This ability to see every variable's value one statement as a time is super powerful for debugging. Use it and save 80% of your development time – really!

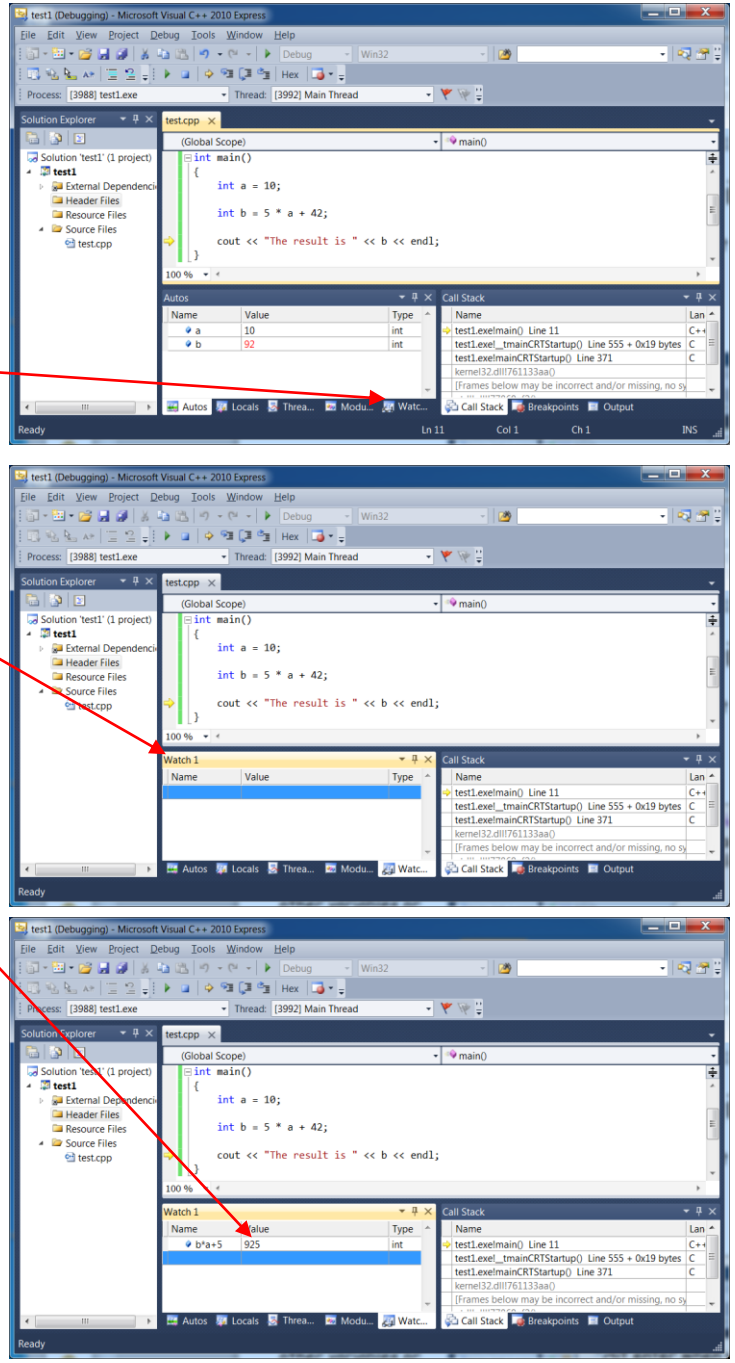






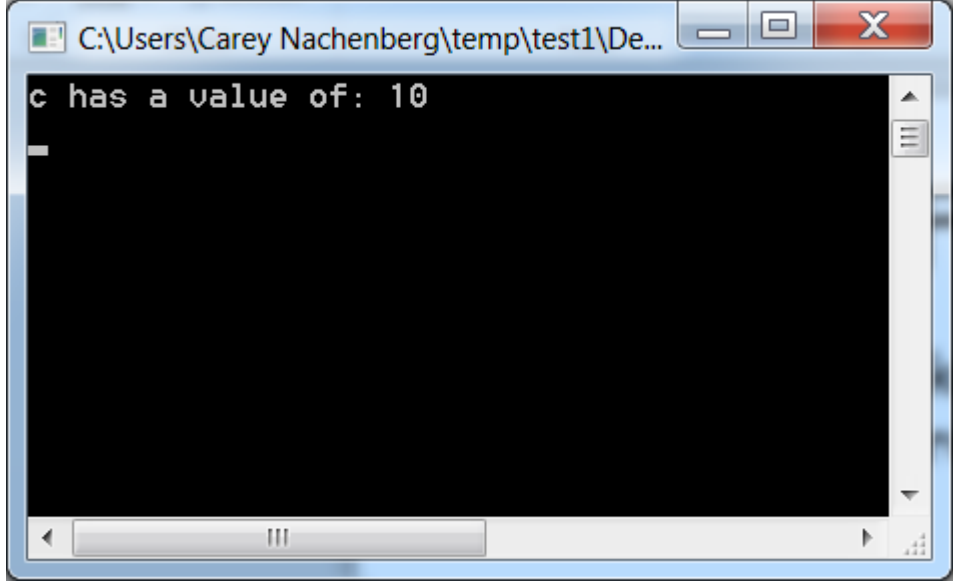
Debugging

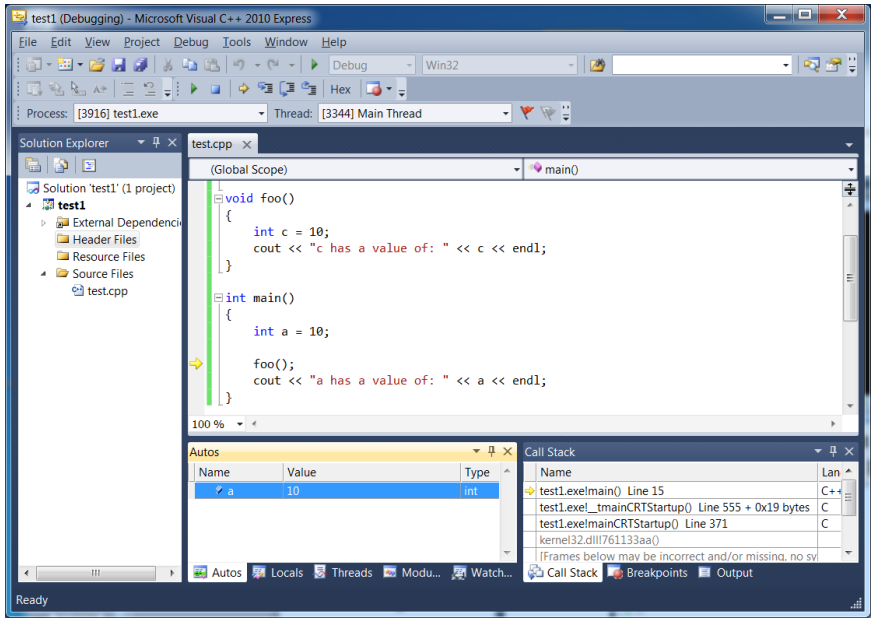
- While you're tracing through your code, you may wish to look at other variables (like member variables) or even evaluate more complex expressions (e.g., 10 * a + 7)
- To do this, click on the Watch tab.
- This will bring up the "Watch" window which allows you to evaluate any variable or expression.
- Click the mouse on the blue line in the Watch window and type in a variable name or expression to watch, e.g., b*a+5. Then hit enter.
- Now the Watch window will display the value of your expression!
- As your code changes the values of variables that would impact the expression (e.g., a or b), the resulting value will change automatically!

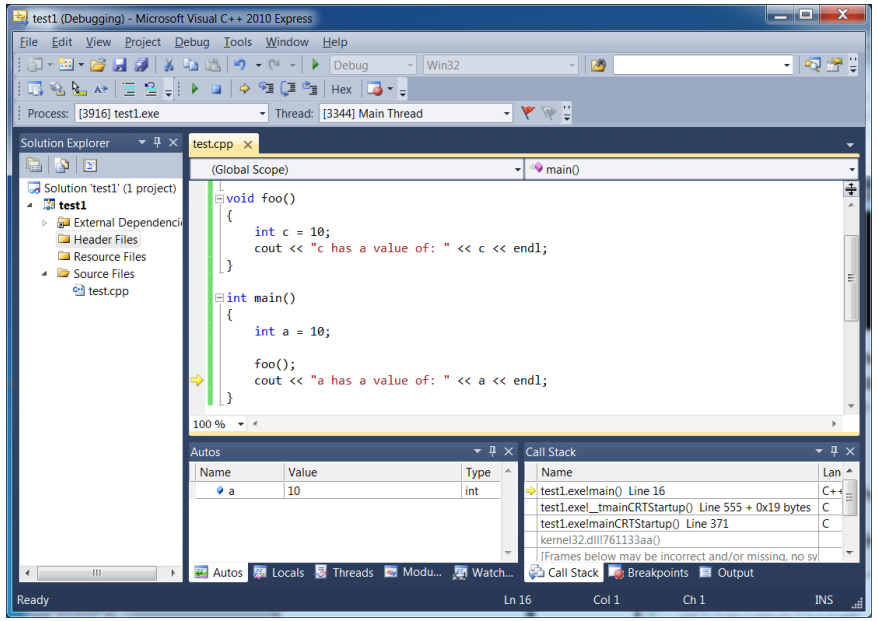


Debugging

- The F10 and F11 keys actually do slightly different things… Let's discuss F10 first. F10 means "TRACE OVER"
- If you hit F10 while on a function call line, Visual Studio will run the entire function in one step. It will trace over all of the statements in that function in one step (even if there are millions of steps). You won't be able to trace line-by-line through the function.
- So F10 lets you quickly run a line of code that calls a function without having to trace through the function.
- Hitting F10 on the foo() line, as shown in the window below, would run the entire foo function from start to finish, producing the output shown in the upper right-hand console window.
- Then Visual Studio would place the cursor on the line following the function call (as shown in the lower right-hand image), where you can continue tracing.

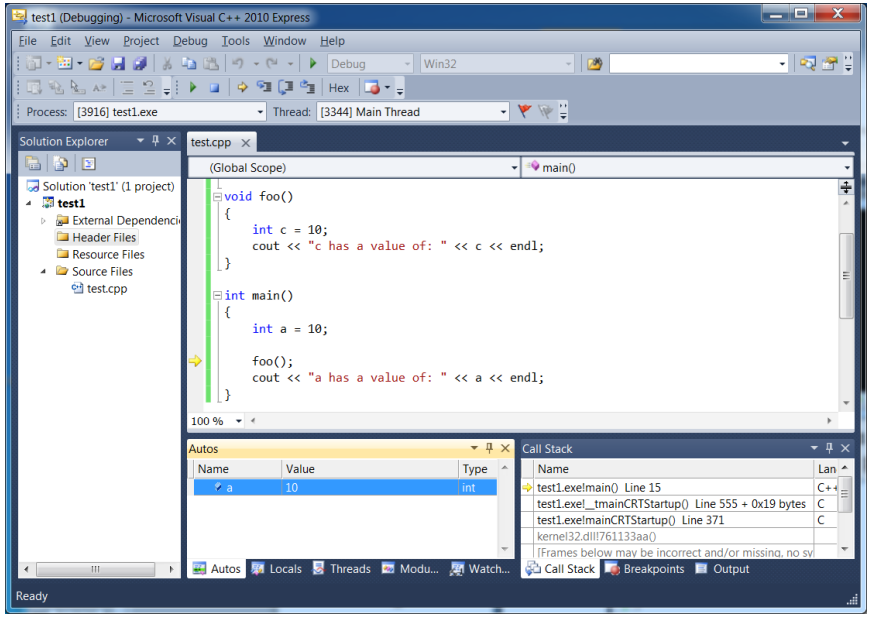


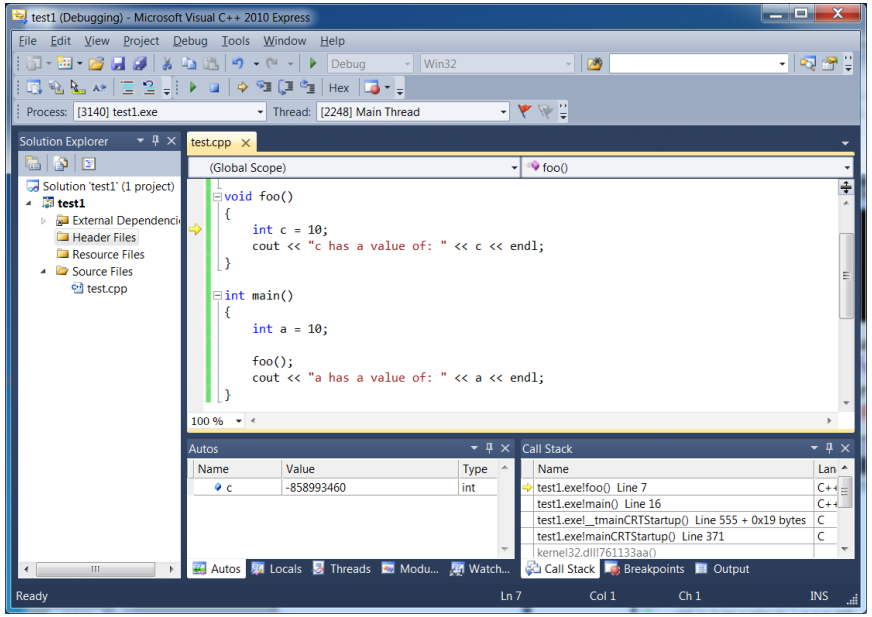




Debugging

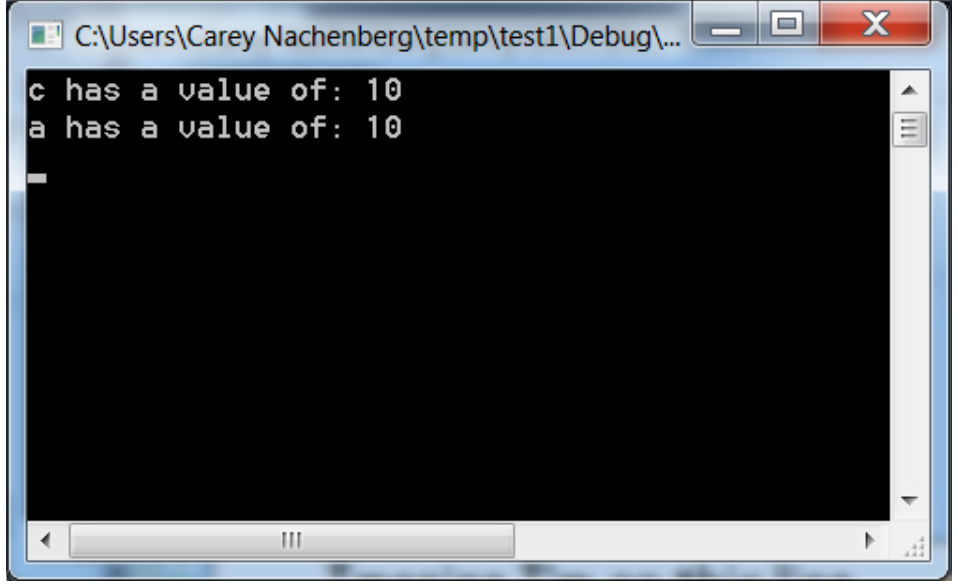
- Let's discuss F11 next. F11 means "TRACE INTO"
- If you hit F11 while on a line that contains a function call, Visual Studio will let you step *into* that function and trace through that function's statements a line at a time.
- So let's say you're debugging your program and are about to call the foo() function (see upper right corner) – look for the yellow arrow next to the green vertical bar.
- If you hit the F11 key, this will take you **into** the foo() function and let you step through its statements a line at a time, as shown in the bottom right (notice the yellow arrow is on the int c = 10; line).
- You can then trace using either F10 or F11 through the foo function. If foo calls another function, you can use F11 to step into that function's logic as well if you like. Or just F10 to step over its code.
- When the foo function finally returns, you can continue tracing at the cout << "a has a value…" line.
- So… use F10 if you want to run a function but skip over the line-by-line debugging. Use F11 if you want to do in-depth debugging of a function call.

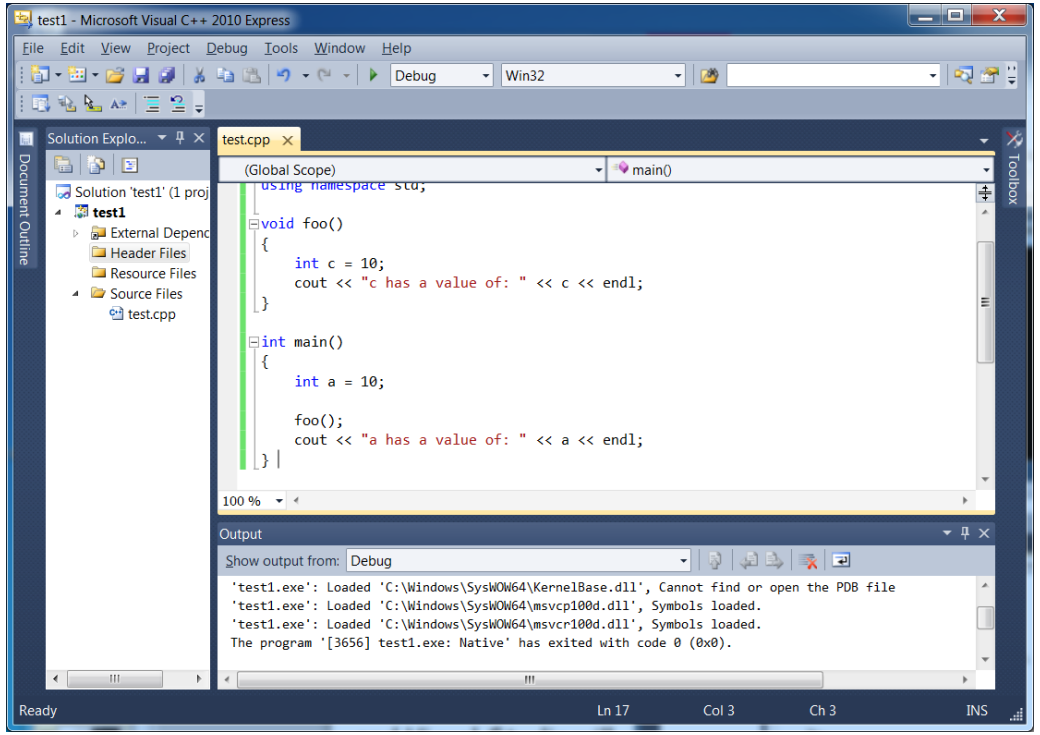




Debugging

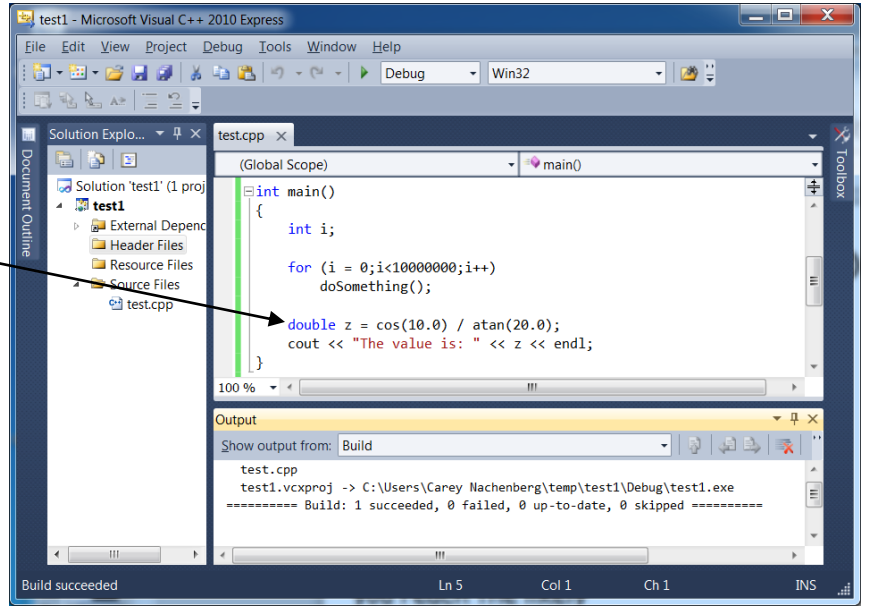
- If at any time while you're tracing line-by-line you want to stop tracing line-by-line…
- And simply let the rest of your program run normally…
- Just hit F5 and Visual Studio will run your program until completion!
- Be careful… your output window will immediately disappear after the program finishes running, so it might look like it's not working.
- If you hit CTRL-F5 instead, that will run your program and stop the terminal window (the black window shown here in the upper left) from disappearing once the program finishes running.

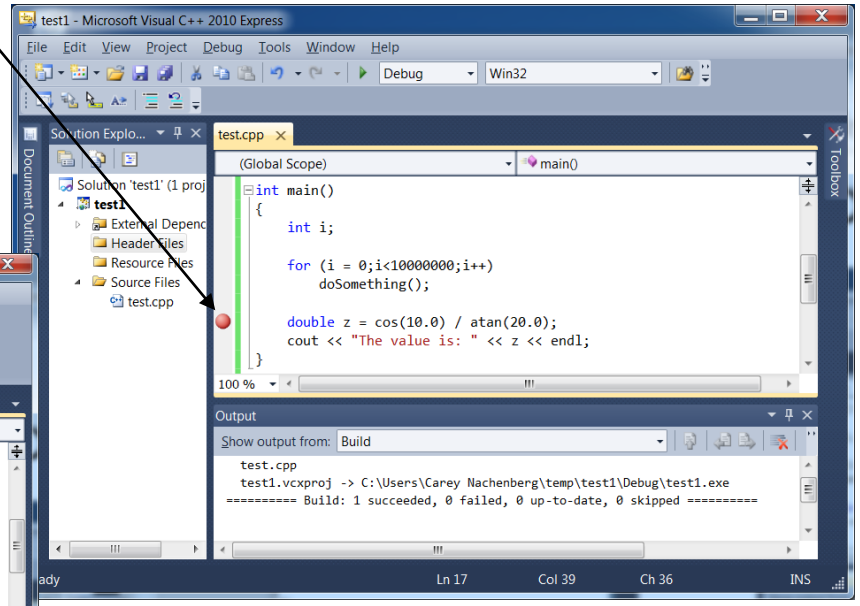


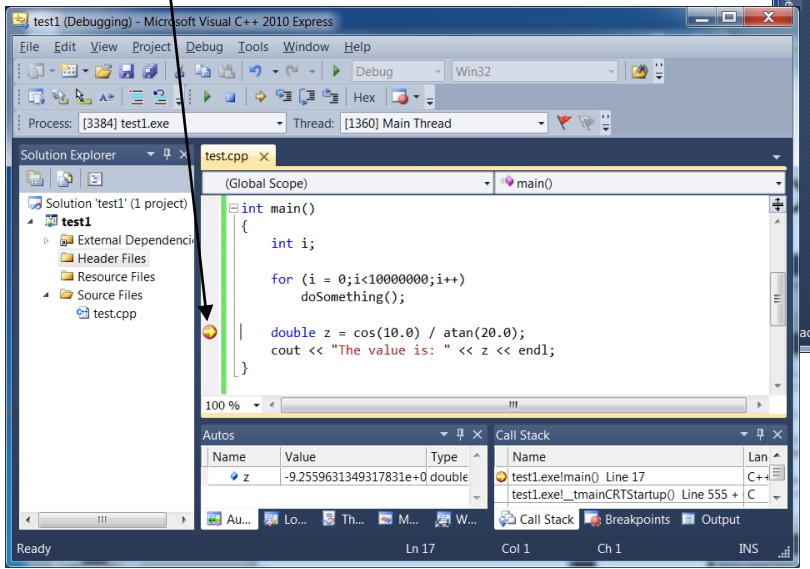


Debugging

- Sometimes you'll want to start debugging deep within your program. Like in the program to the left – there's a 10 million iteration loop before the important computation that divides cos/atan that I want to debug.
- You don't want to step through millions of lines to reach the likely location of a bug. What should you do?
- Answer: Add a breakpoint to your program.
- A breakpoint is a tag that you add to a line in your program that tells the debugger to stop when it hits that line.
- To add a breakpoint, click the mouse on the line where you'd like to start debugging and hit the F9 key. You'll see a red ball show up next to the line.
- Then just hit the F5 key to run your program from the start until it reaches that line! Your program will run potentially millions of instructions in seconds until it hits the breakpoint.
- Then the debugger stops the program, shows you the yellow arrow, and lets you trace line-by-line using F10 and F11 just like before.







- Or if you like after debugging a few lines, you can hit F5 again and your program will run until it hits the breakpoint again.
- Oh, and you can add as many breakpoints as you like… Not just one!

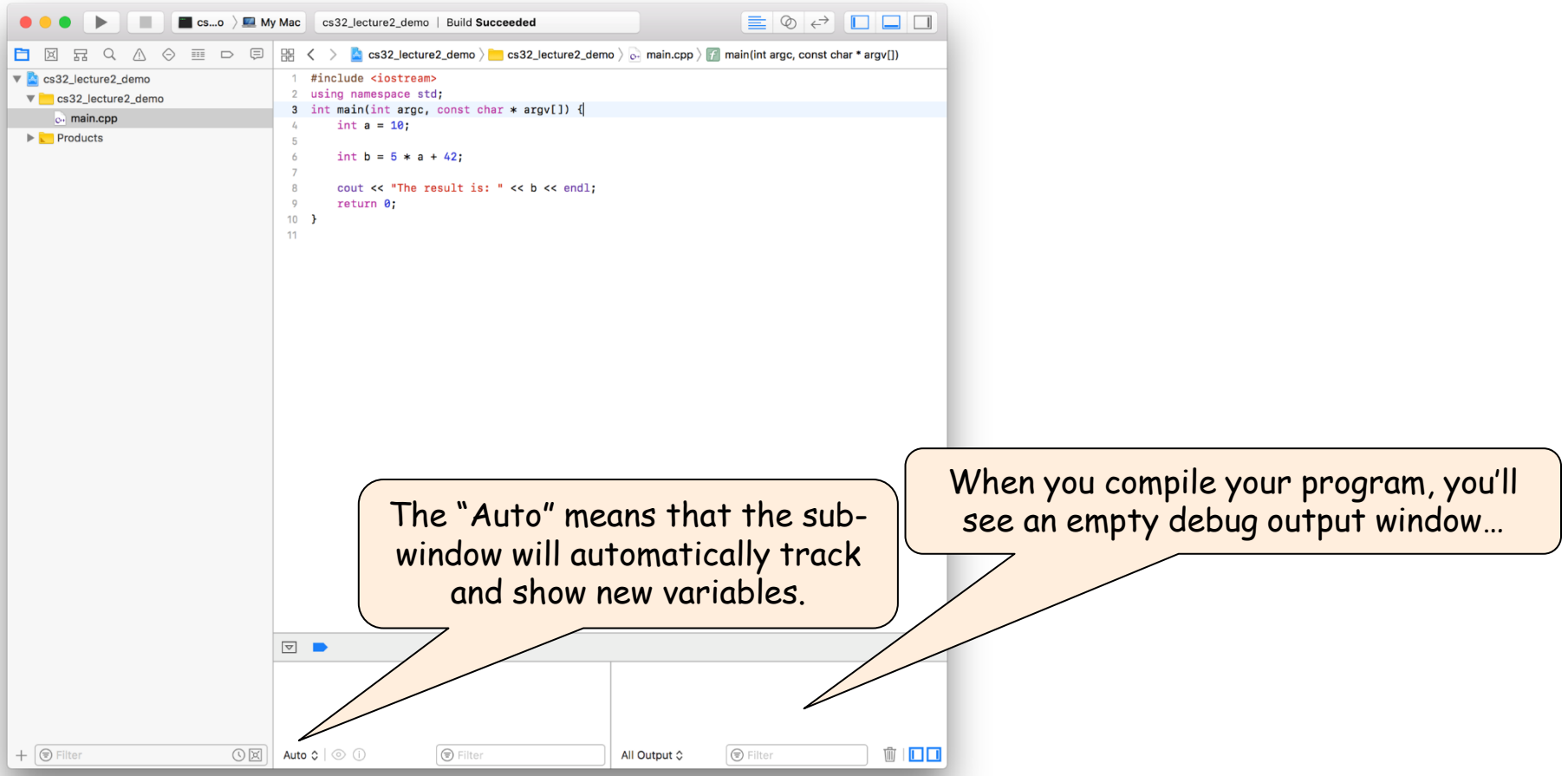
Debugging with Xcode

The **SUPERIOR** IDE ☺

(Xcode debugging slides graciously created by Devan Dutta!)

Debugging with Xcode

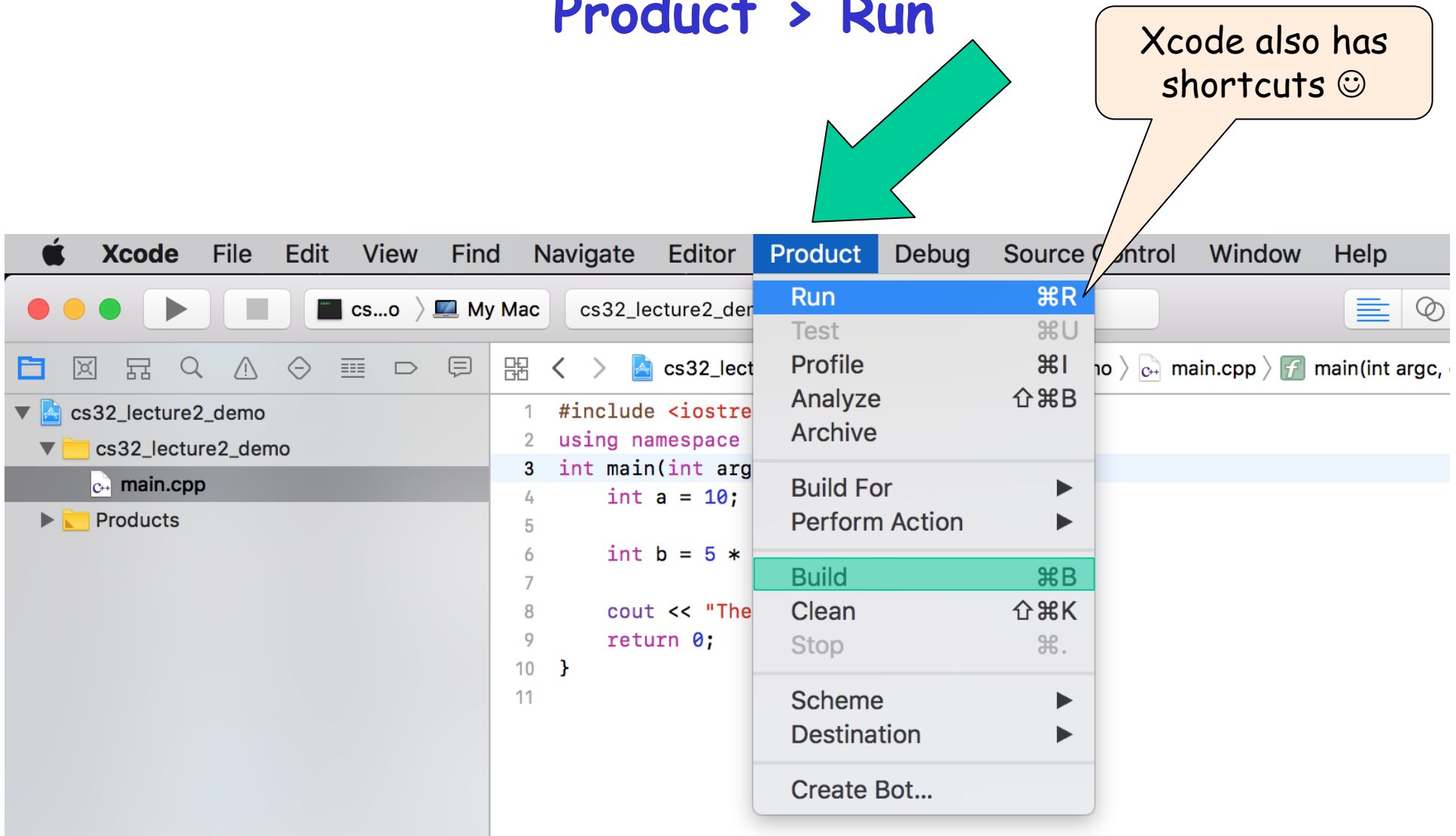
This is Xcode's code editor. Xcode is an **IDE** created by **Apple**, specifically for iOS, macOS, tvOS, and watchOS development, but it also comes fully loaded with **C and C++ support**.



Debugging with Xcode

You can run your program from start to finish by going to:

Product > Run



Debugging with Xcode

You can also step through your program, one line at a time.

Let's insert a breakpoint at line 3.

```
1  #include <iostream>
2  using namespace std;
3  int main(int argc, const char * argv[]) {
4      int a = 10;
5
6      int b = 5 * a + 42;
7
8      cout << "The result is: " << b << endl;
9      return 0;
10 }
11
```

Just tap on the gray space next to the 3.

Debugging with Xcode

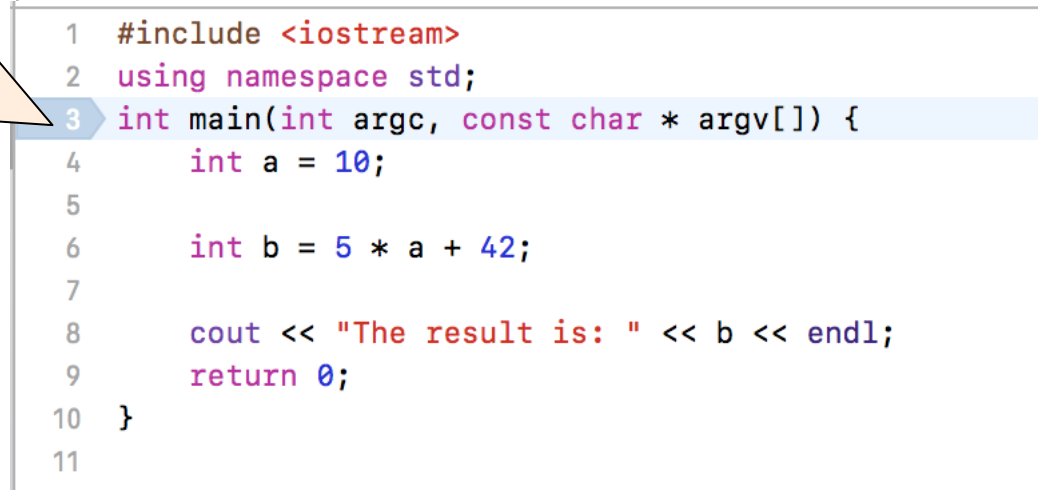
You can also step through your program, one line at a time.

Let's insert a breakpoint at line 3.

The blue arrow on line 3 shows that our breakpoint is **active**.

If we click on the breakpoint again, we **deactivate** it, but Xcode remembers that we had a breakpoint there and will let us turn it back on. Neat!

```
1  #include <iostream>
2  using namespace std;
3  int main(int argc, const char * argv[]) {
4      int a = 10;
5
6      int b = 5 * a + 42;
7
8      cout << "The result is: " << b << endl;
9      return 0;
10 }
```



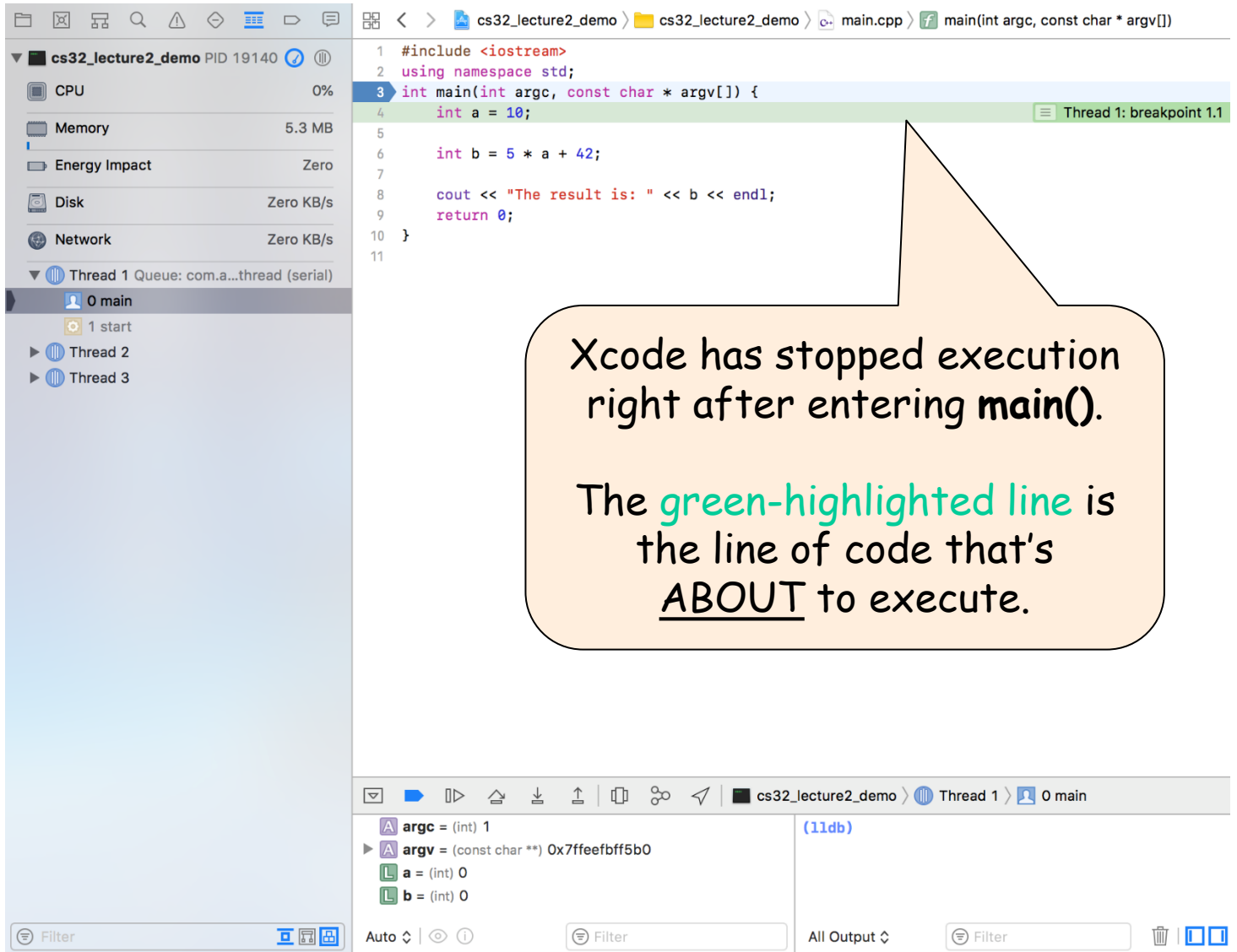
Debugging with Xcode

Because we set a breakpoint at main(), we can debug 1 line at a time.

This screenshot to the right shows what an Xcode debugging session looks like.

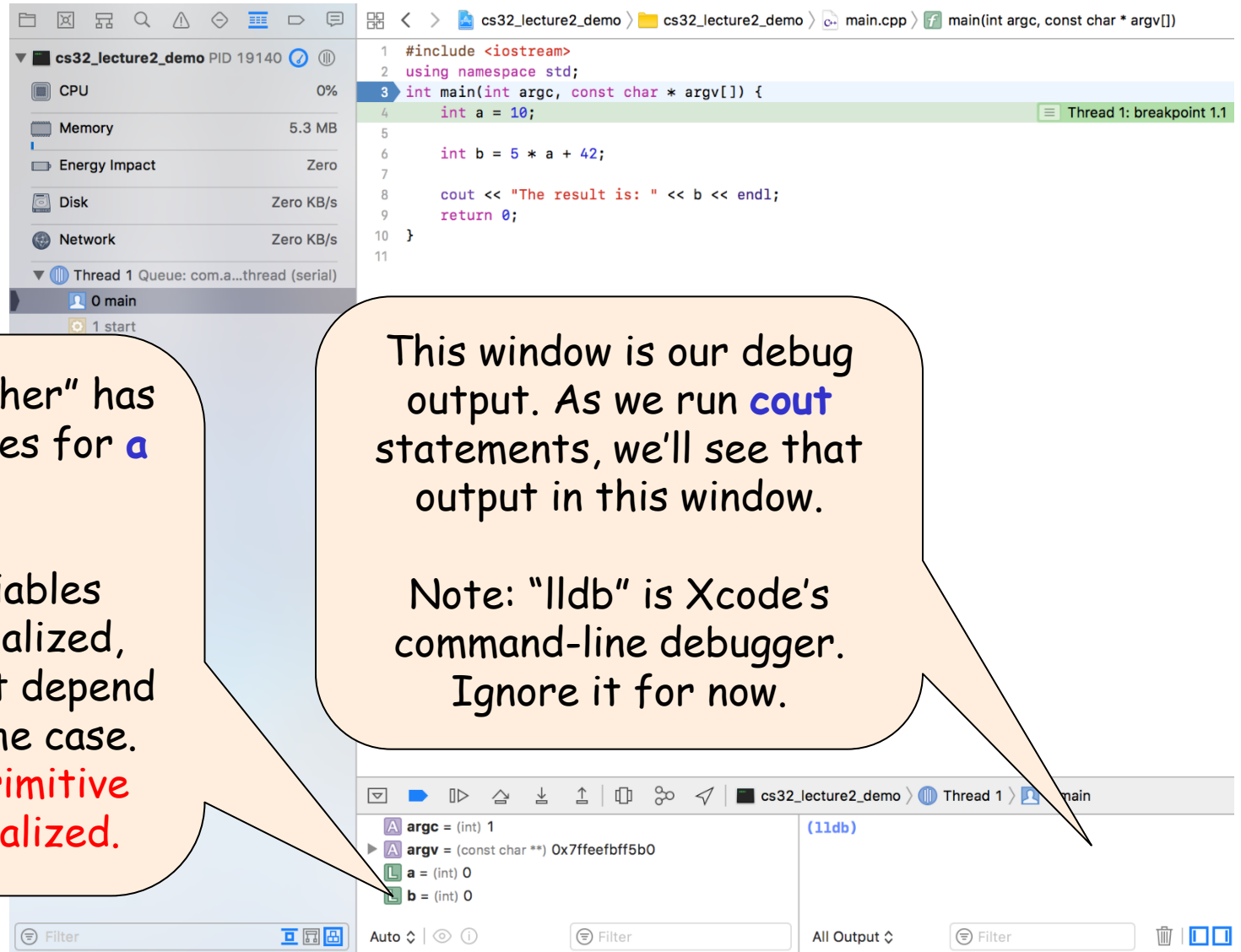
The left pane is showing resource usage.

The right pane is the code being run.



Debugging with Xcode

Because we set a breakpoint at main(), we can debug 1 line at a time.



Our variable "watcher" has also indicated values for **a** and **b**.

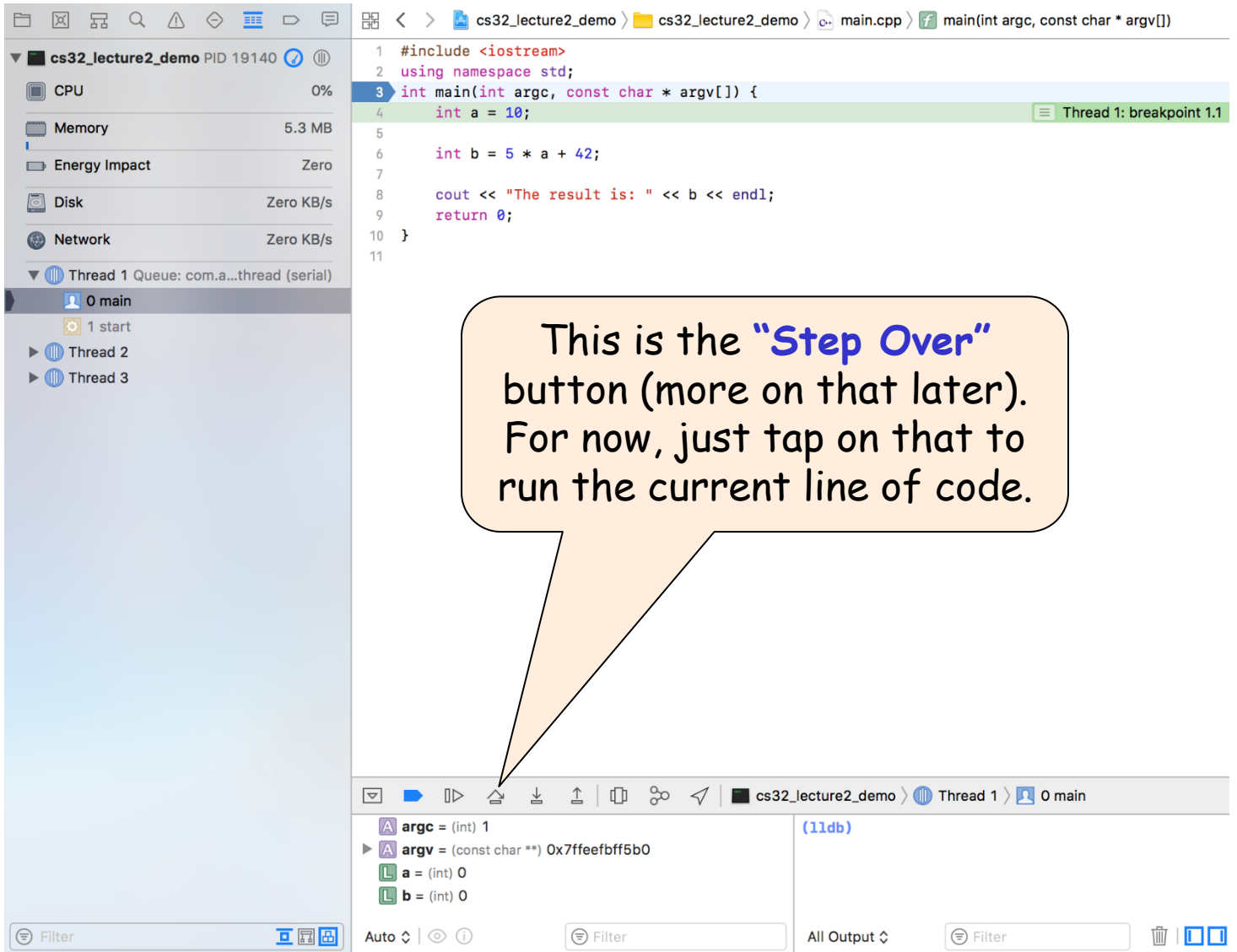
The a and b variables appear to be initialized, but in C++ you can't depend upon that to be the case. Always assume primitive values are uninitialized.

This window is our debug output. As we run **cout** statements, we'll see that output in this window.

Note: "lldb" is Xcode's command-line debugger. Ignore it for now.

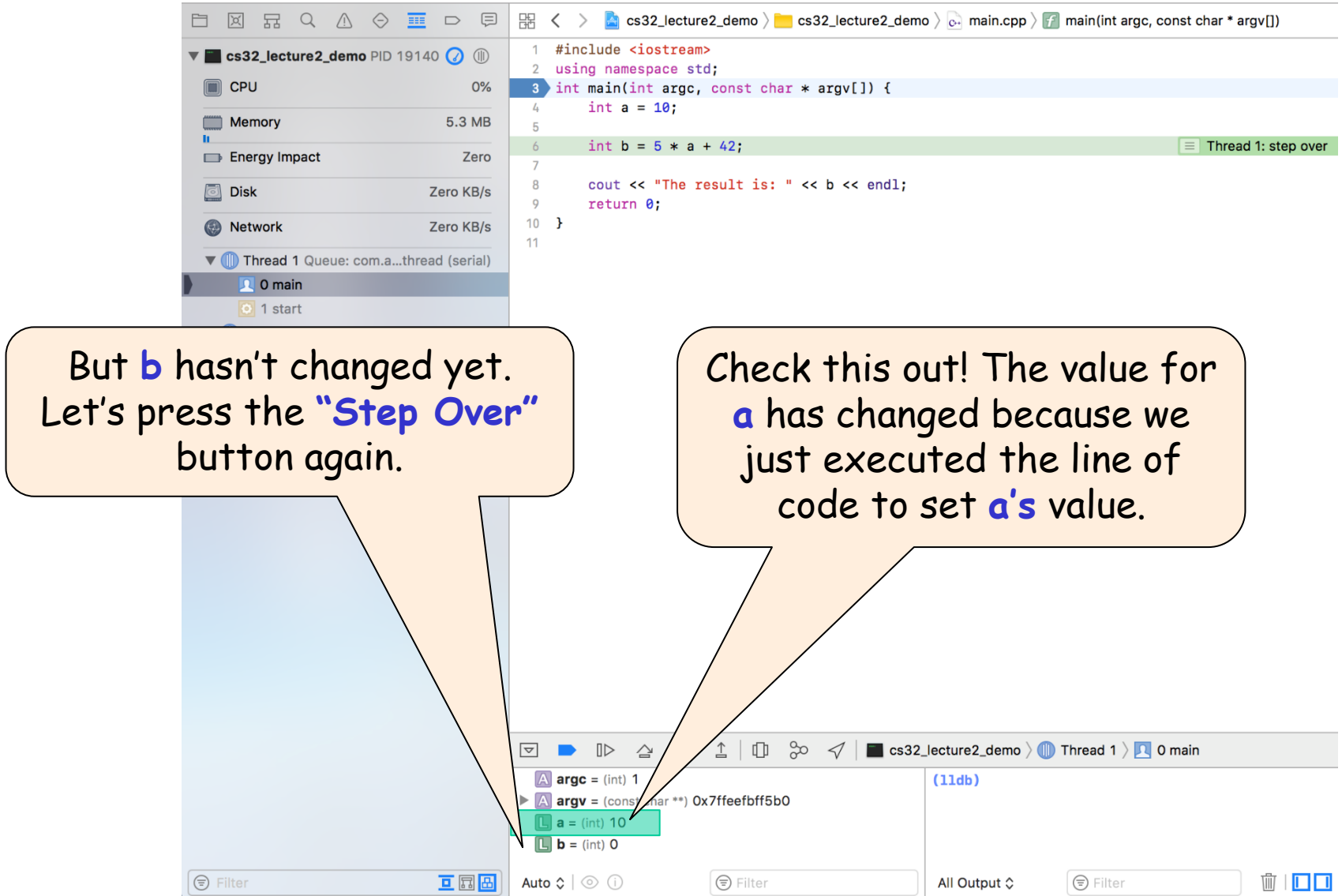
Debugging with Xcode

Let's run the current line of code.



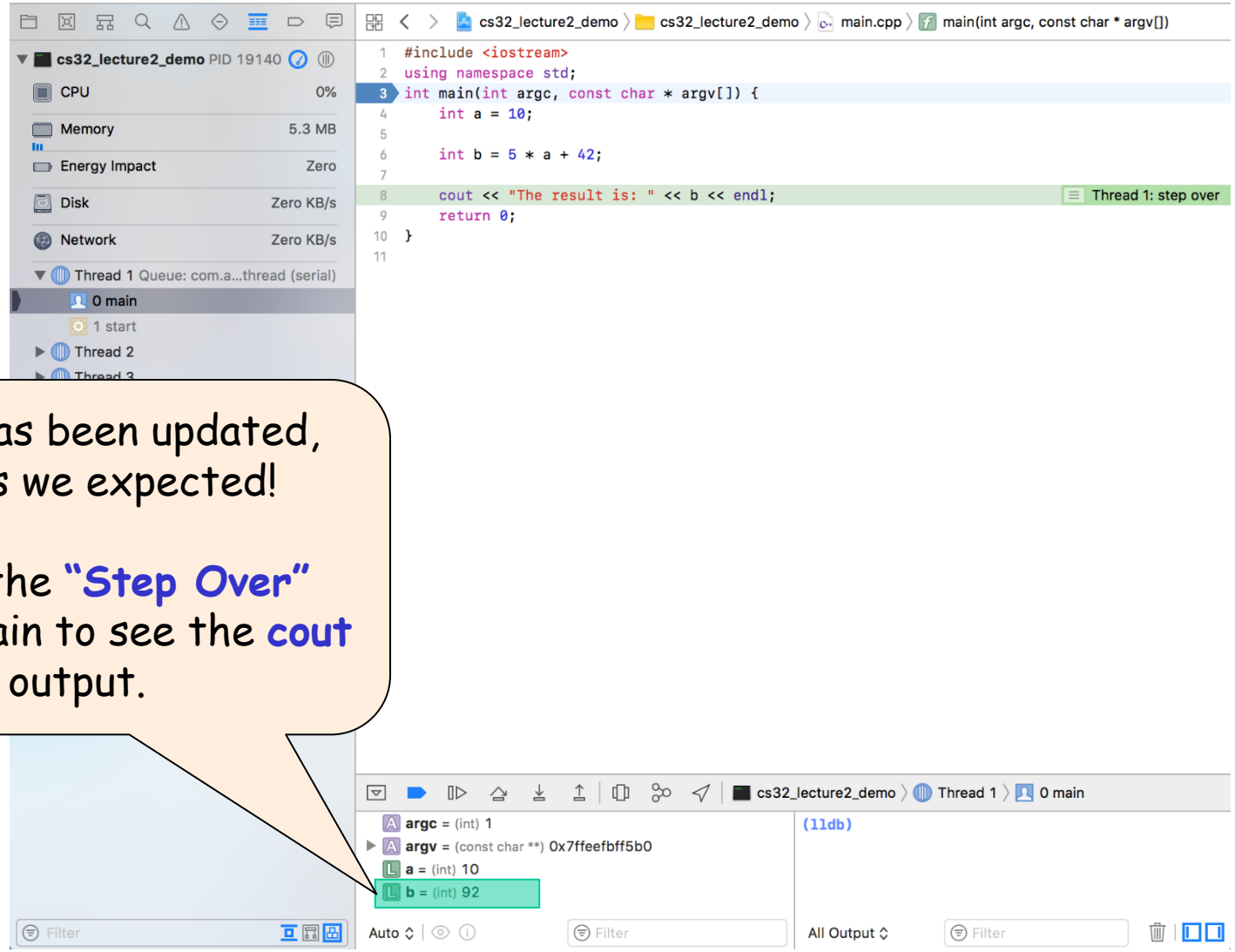
Debugging with Xcode

Let's run the current line of code.



Debugging with Xcode

Let's run the current line of code.

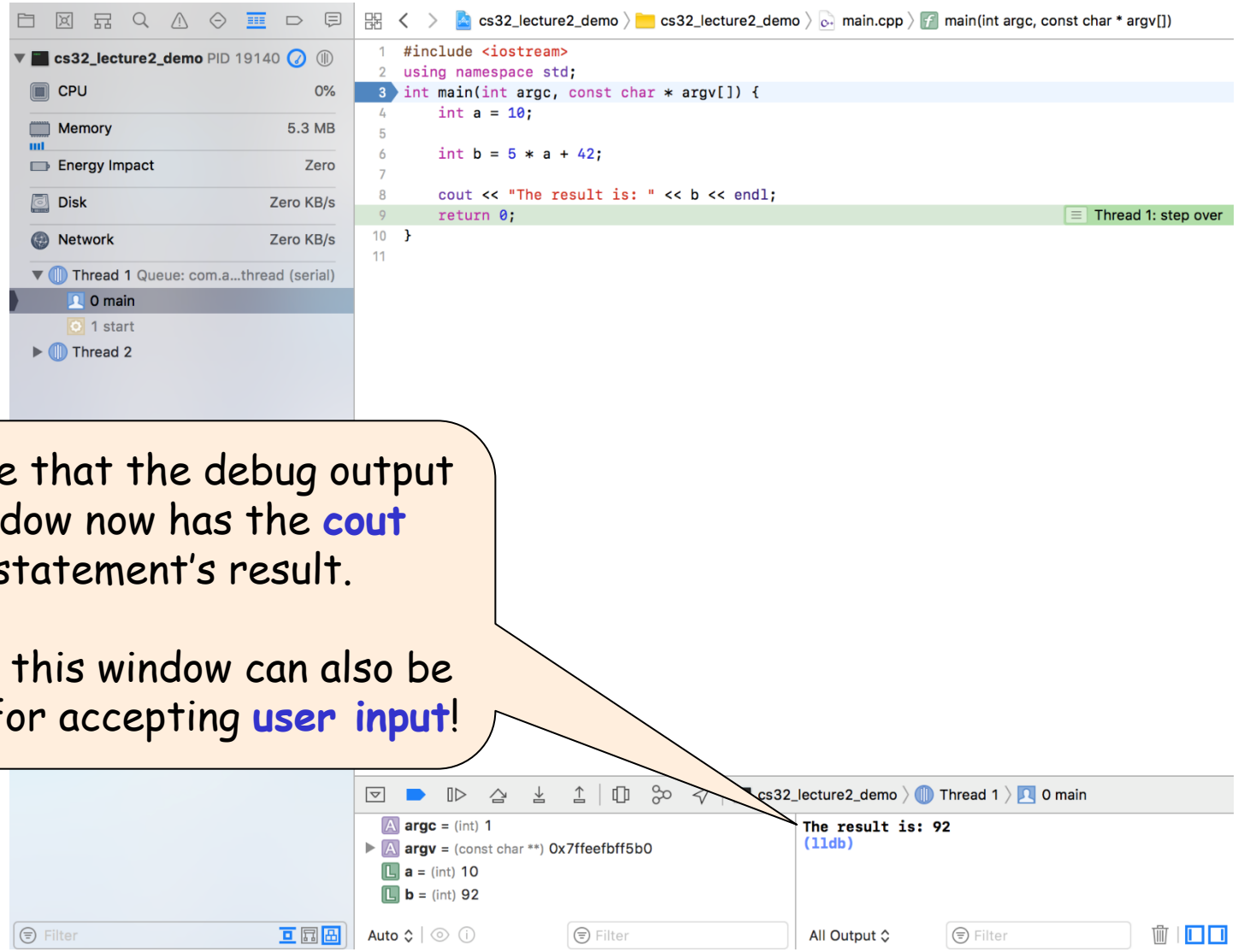


Now **b** has been updated, just as we expected!

Tap on the **"Step Over"** button again to see the **cout** output.

Debugging with Xcode

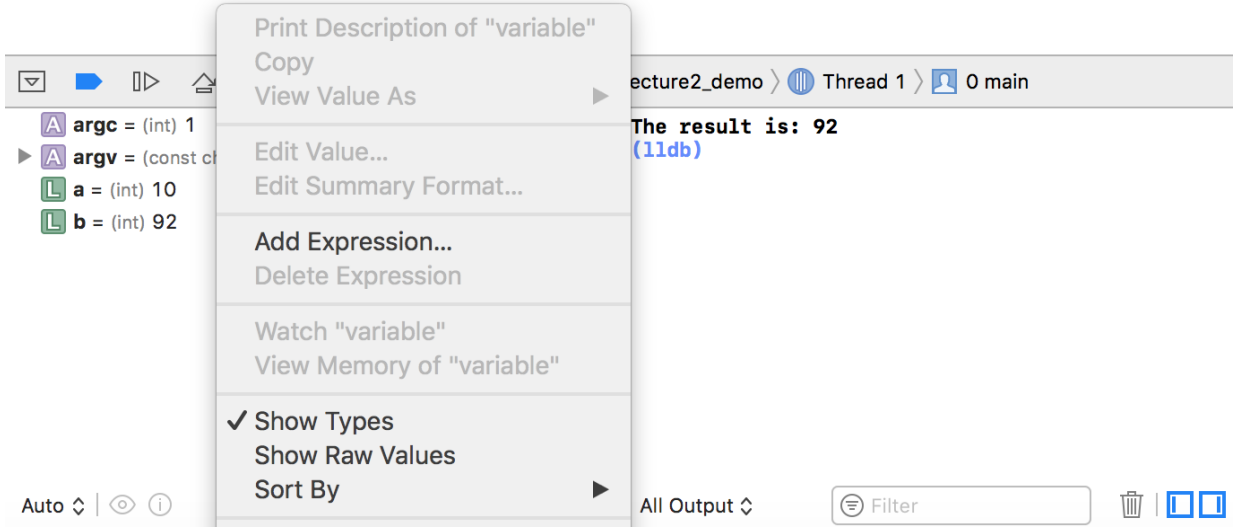
Let's run the current line of code.



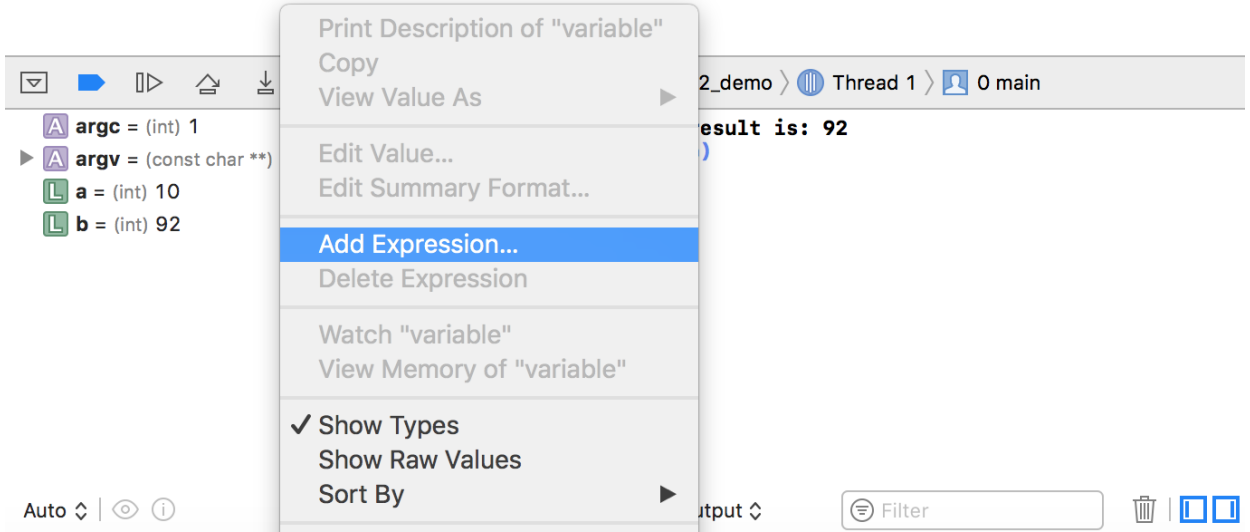
Debugging with Xcode

Let's use the **Watch** window to add an expression.

Right-Click anywhere in the **Watch** window.

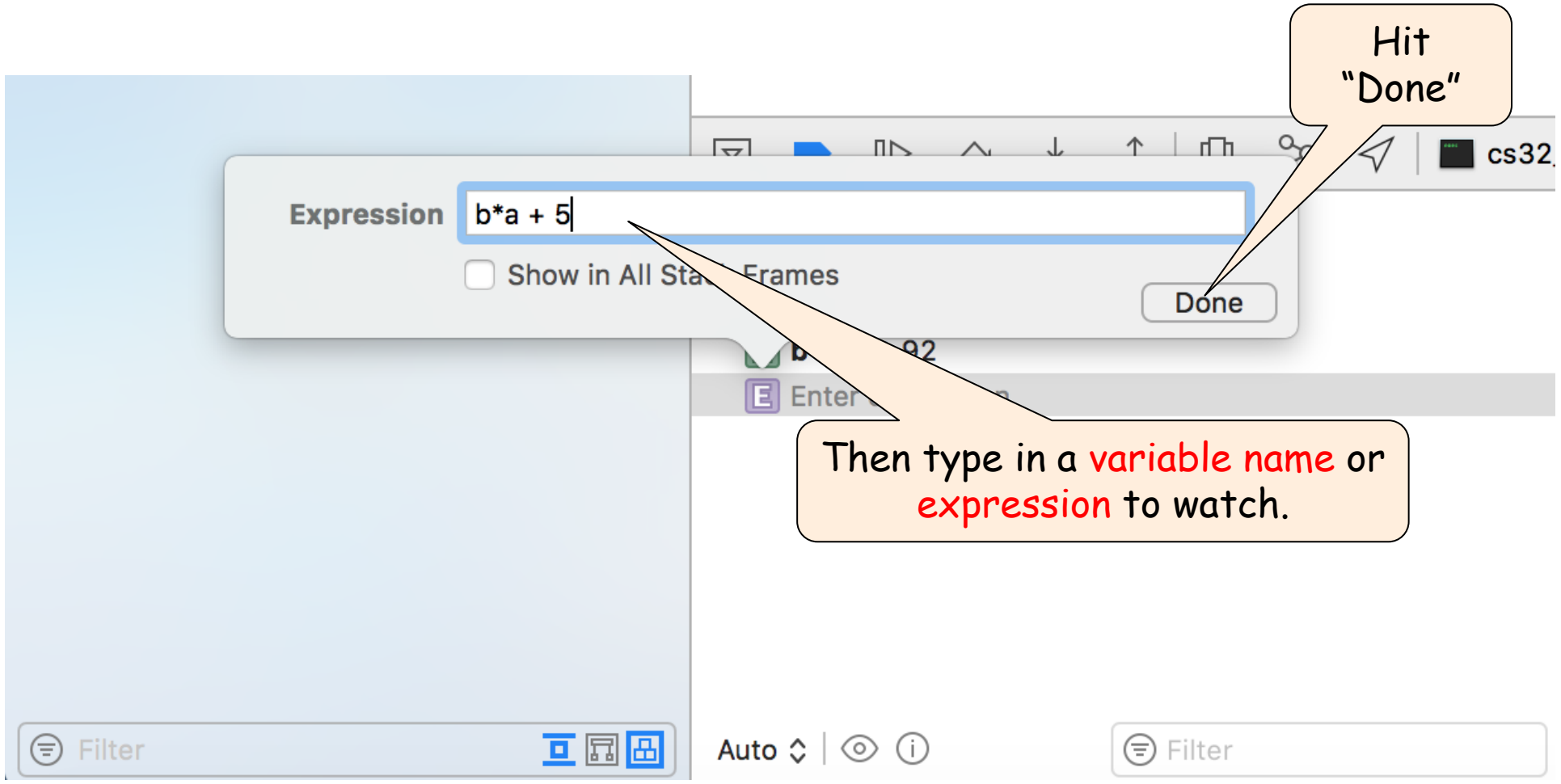


Now click on **Add Expression...**



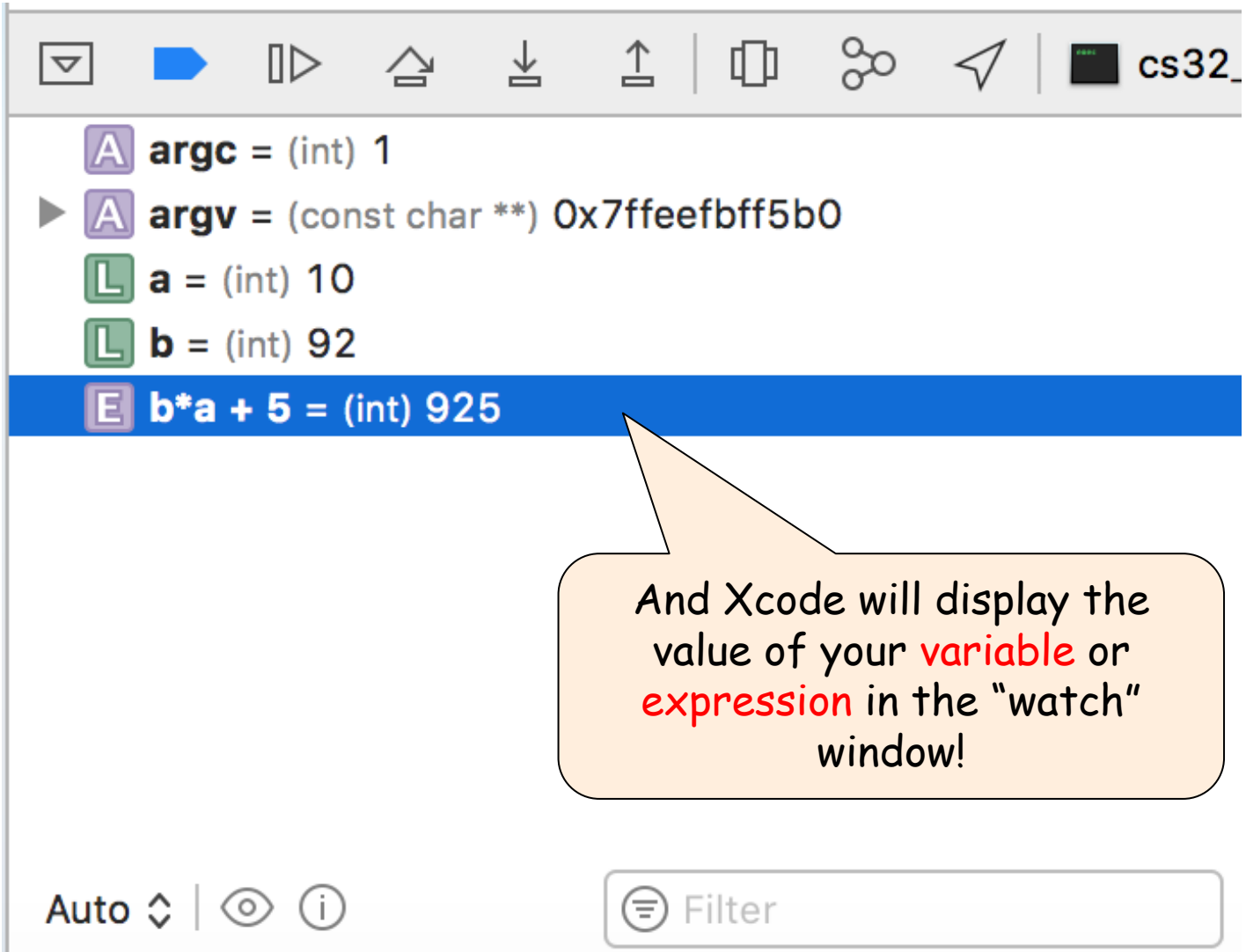
Debugging with Xcode

Add an expression to watch



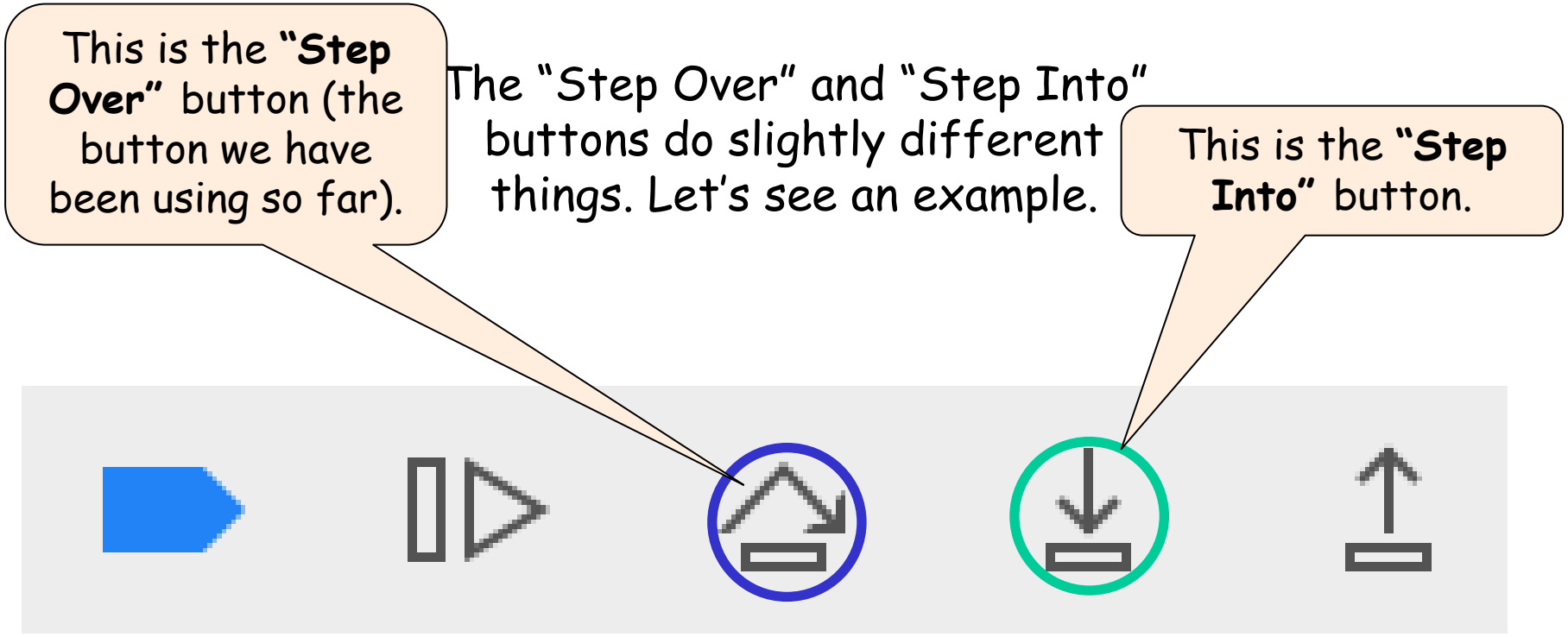
Debugging with Xcode

Add an expression to watch

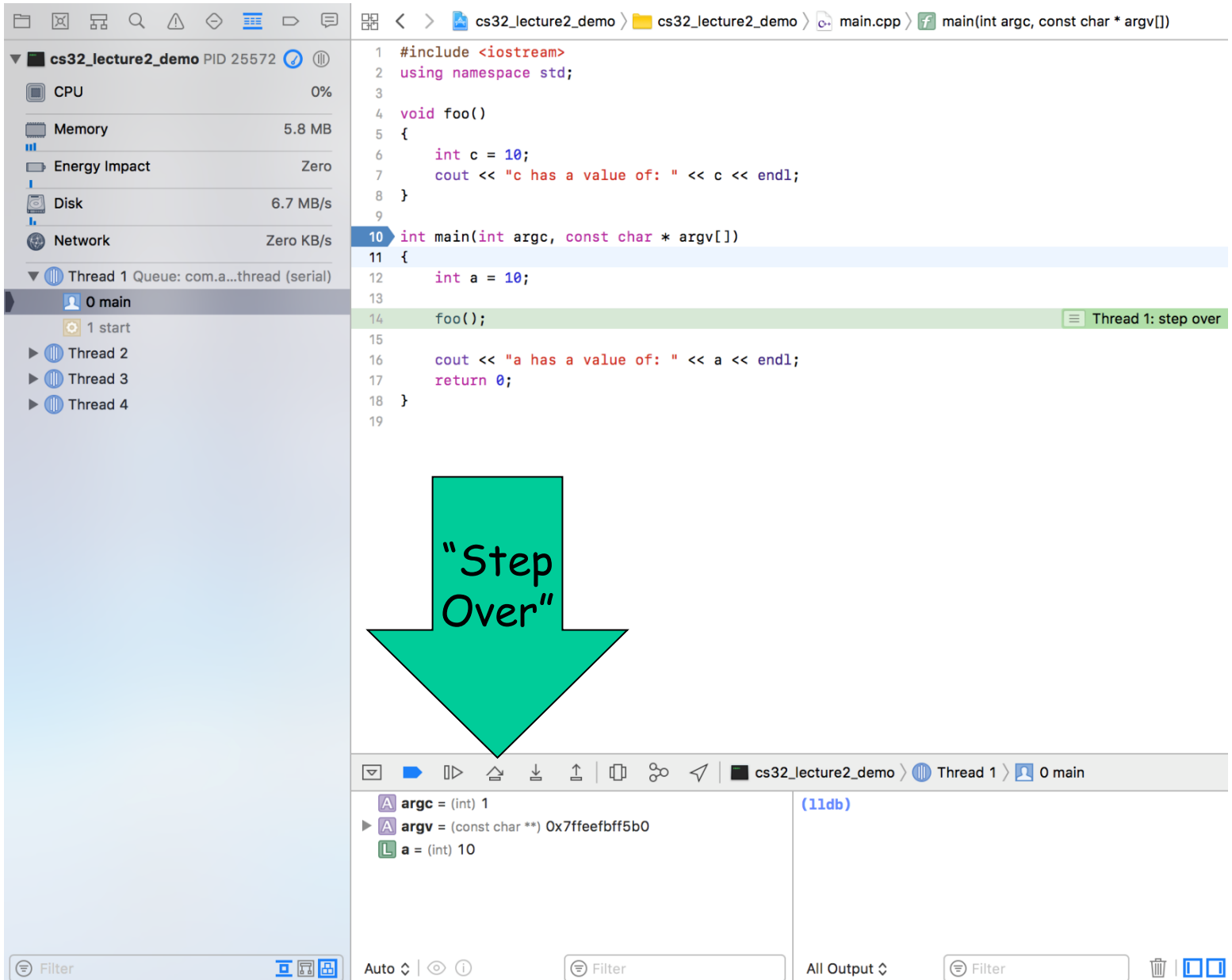


Debugging with Xcode

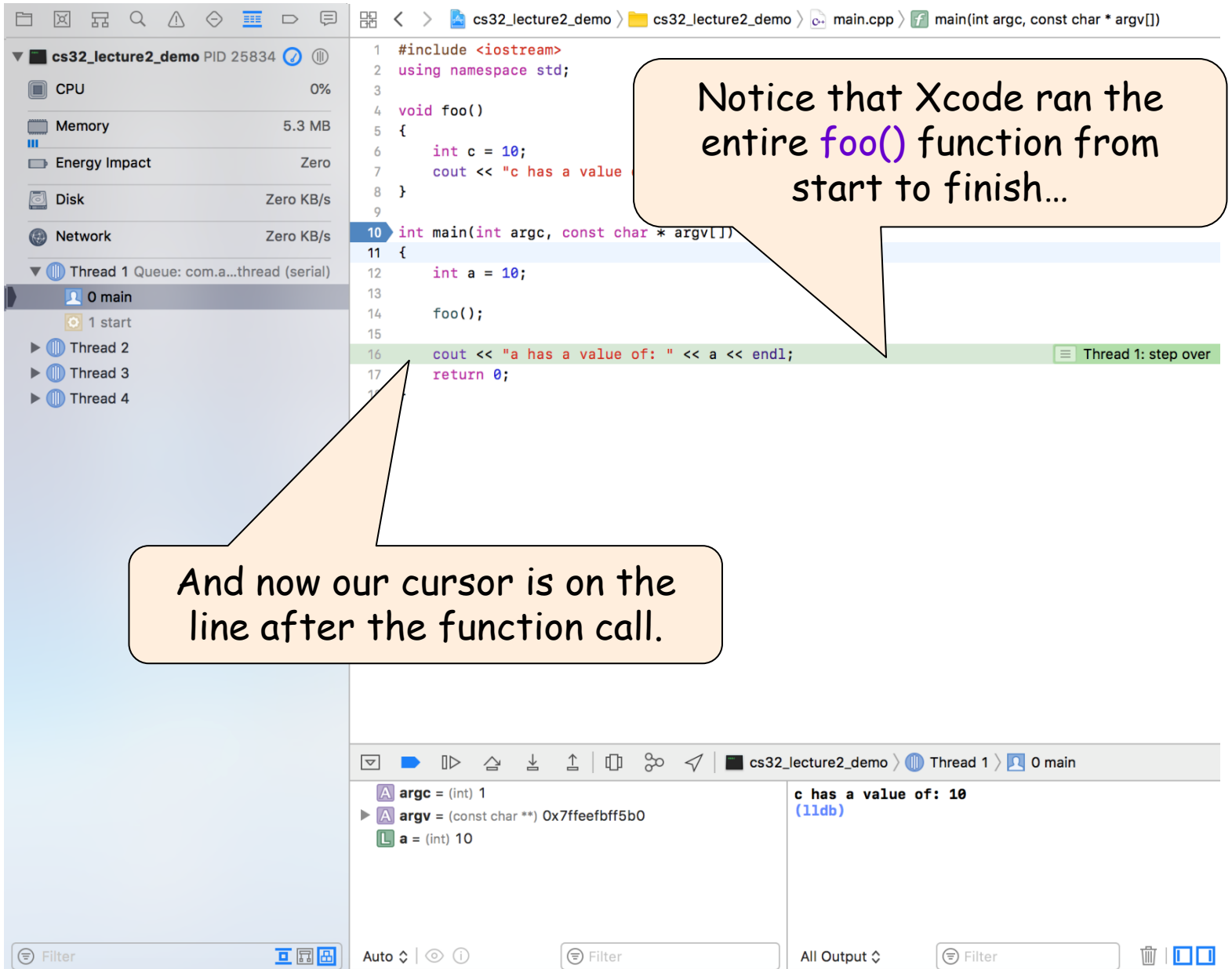
Let's revisit the "Step Over" button. Note the other button right next to it.



Debugging with Xcode



Debugging with Xcode

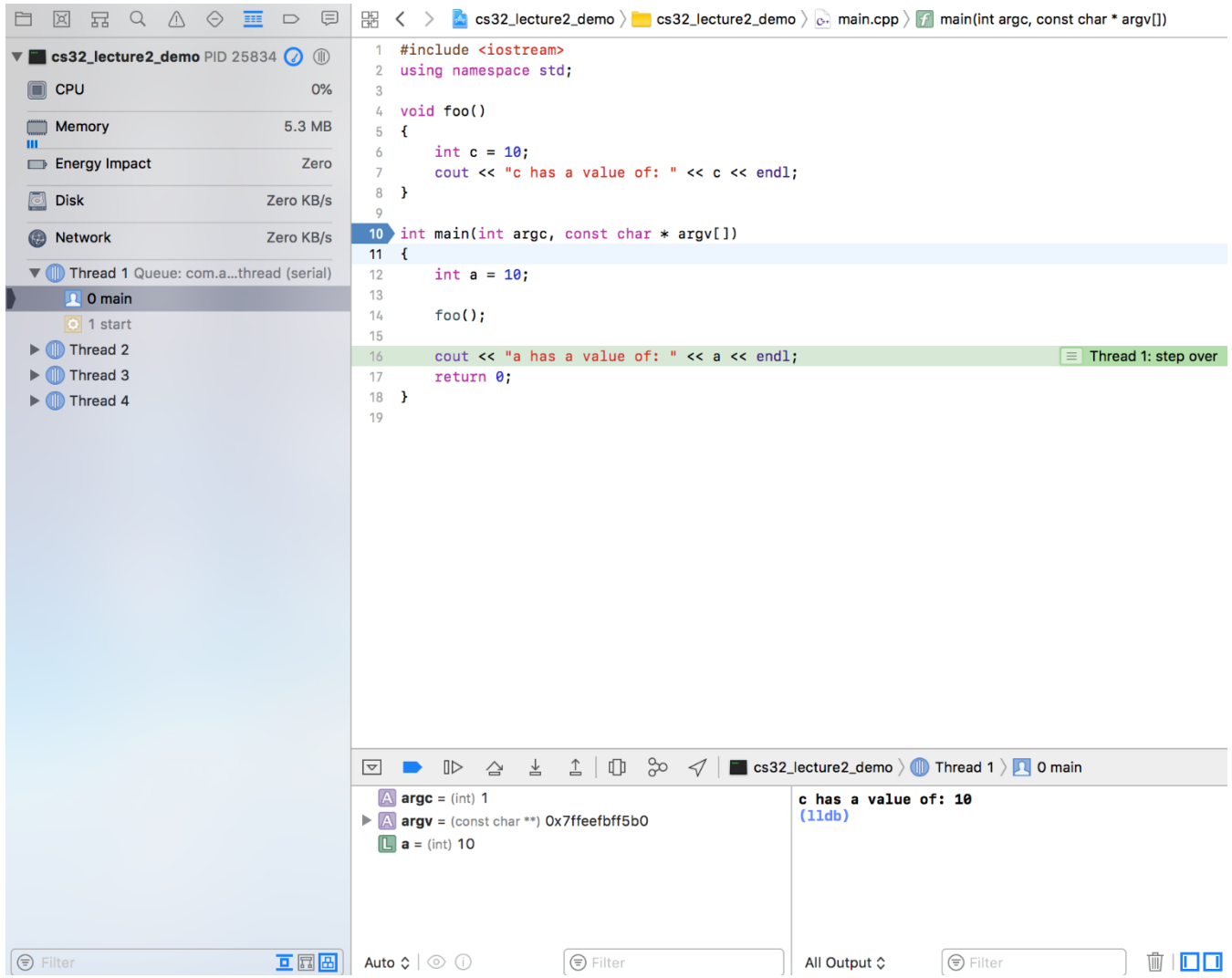


Debugging with Xcode

Summary:

If you hit "Step Over" while on a function call line, Xcode will run the entire function in one step. You won't be able to trace line-by-line through it.

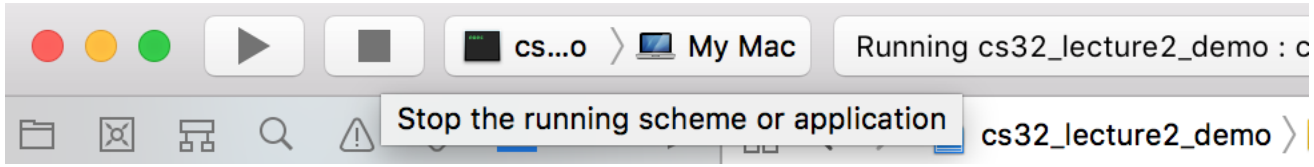
So "Step Over" lets you quickly run a line of code without delving into the details.



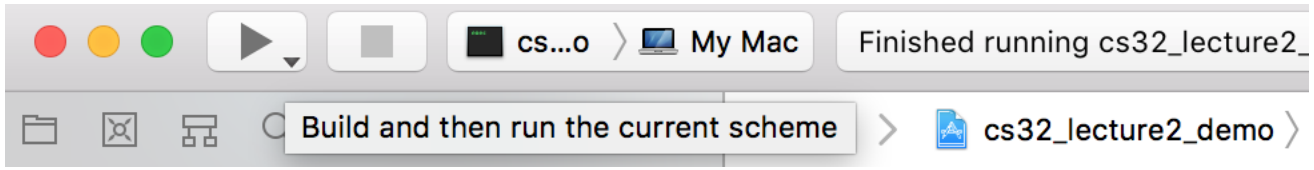
Debugging with Xcode

Let's restart the debugger and use the **"Step Into"** button instead of **"Step Over"**.

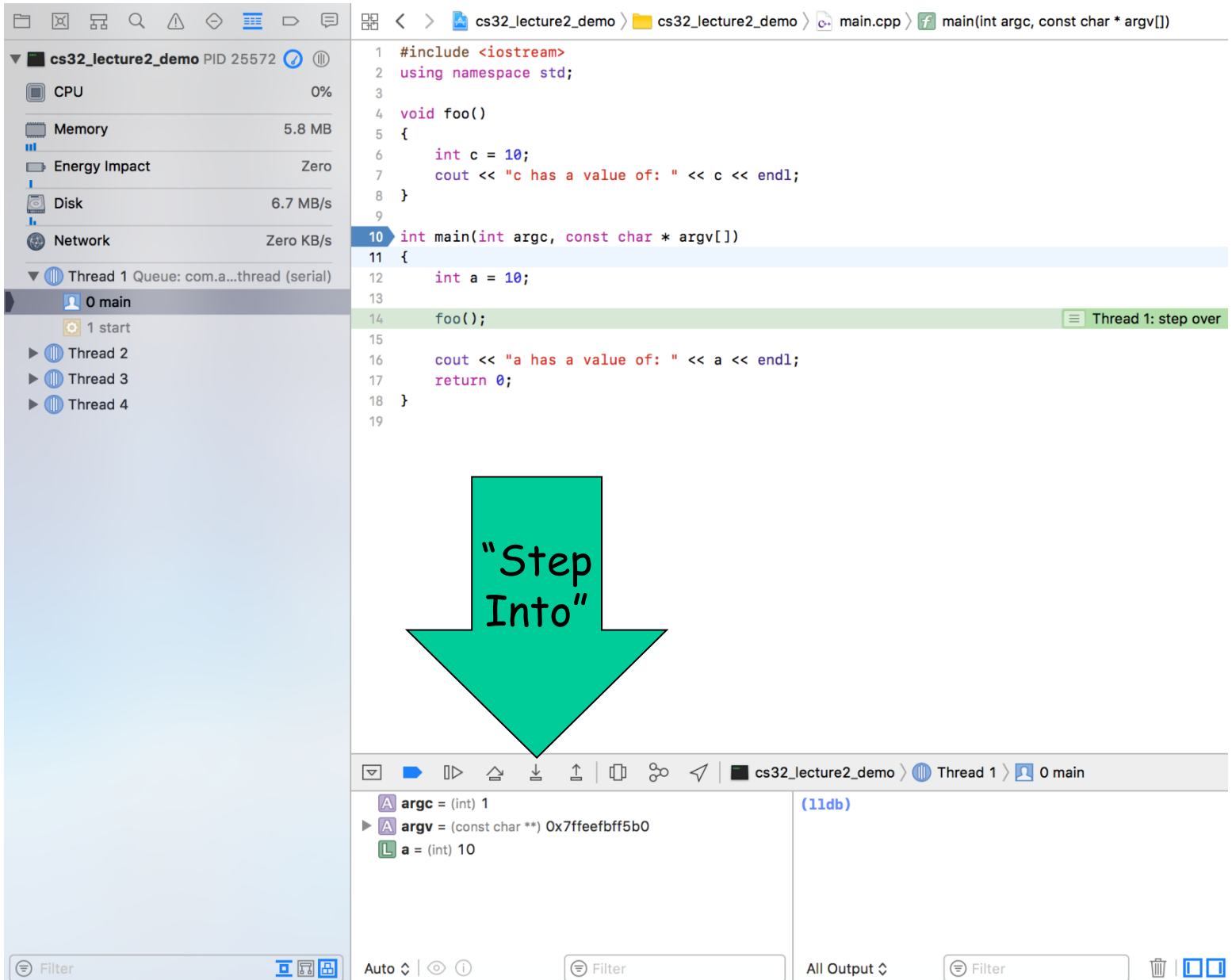
Just tap on the **Stop** button to stop the current debugging session.



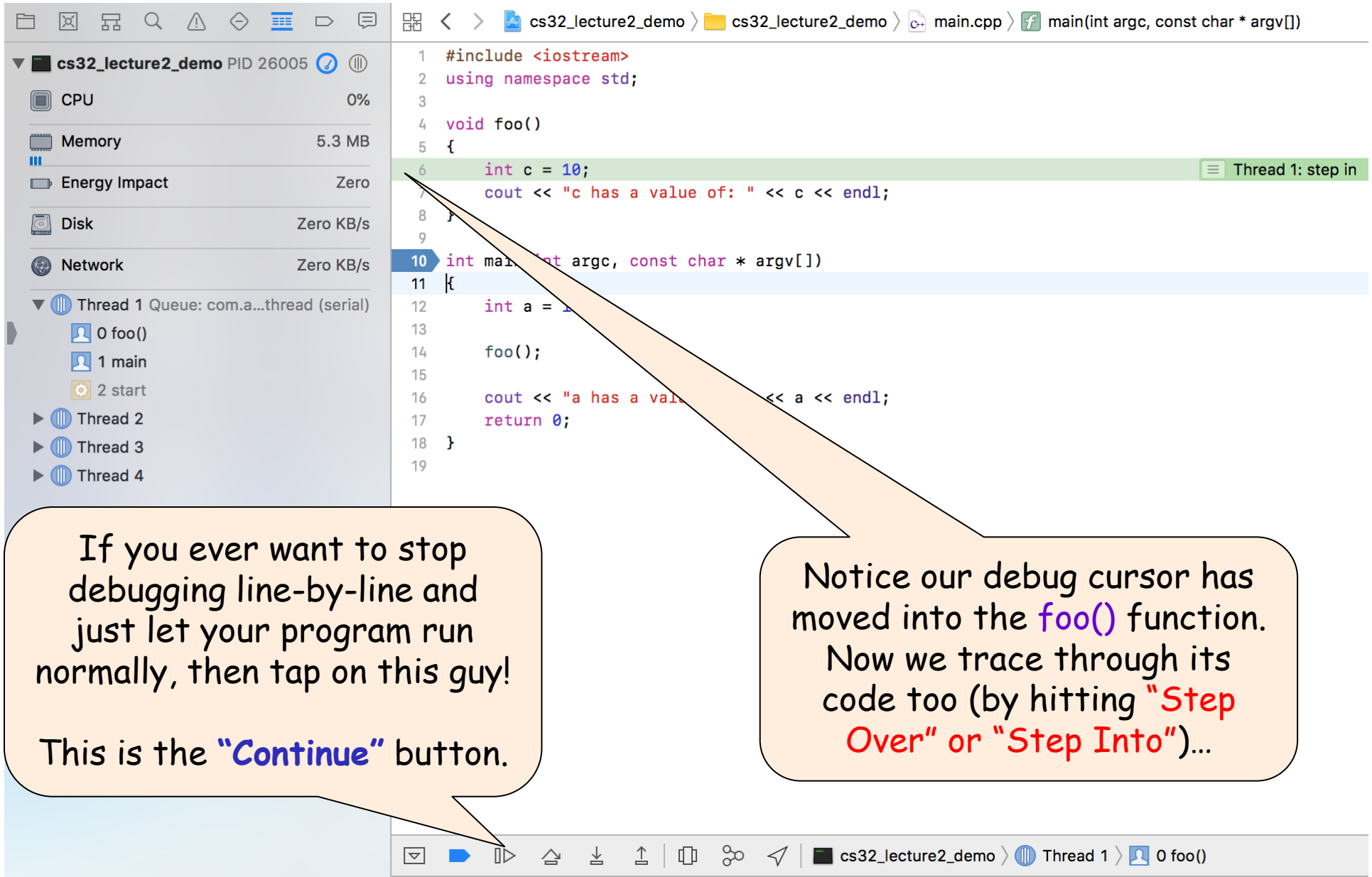
Then tap on the **Run** button to re-build and run the code.



Debugging with Xcode



Debugging with Xcode



Debugging with Xcode

Quick note on breakpoints:

We already used breakpoints in our debugging examples, but there's 1 case where you **really** want to use them.

```
10
11  int main(int argc, const char * argv[])
12  {
13
14      foo();
15
16      for (int i=0; i<1000000; i++)
17      {
18          cout << "I'm wasting my time debugging a for loop" << endl;
19      }

        double z = cos(10.0) / atan(20.0);

        cout << "z has a value of: " << z << endl;
        return 0;
25  }
```

Do I really need to step through this for loop?! That would take years!!!!

What if I want to start debugging at this line?

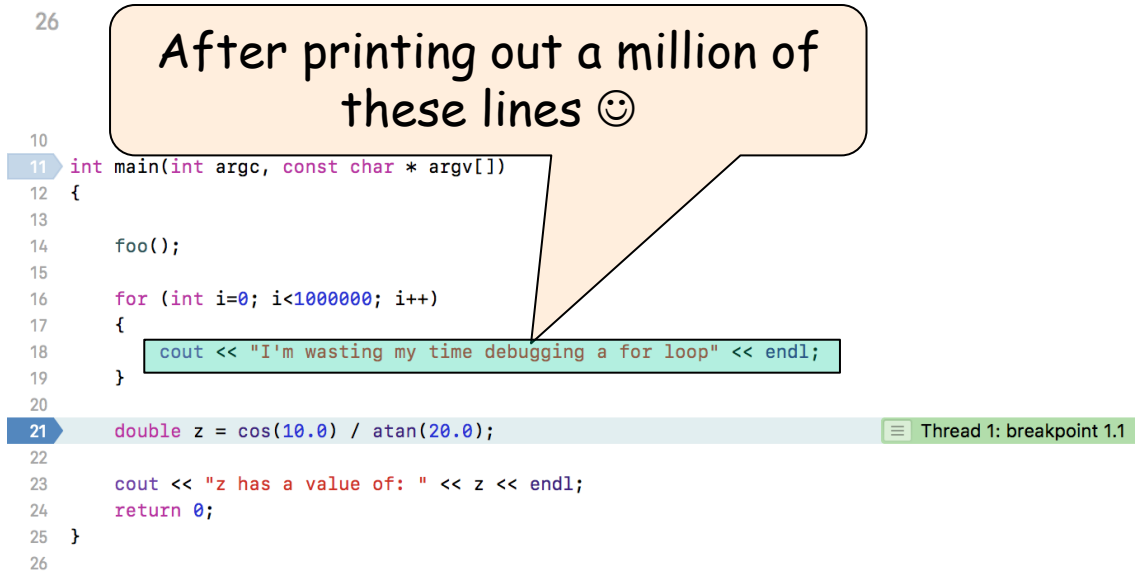
Debugging with Xcode

Debugging with Xcode

Just add a breakpoint!

After pressing **Run**, Xcode will stop at that line…

```
10
11  int main(int argc, const char * argv[])
12  {
13
14      foo();
15
16      for (int i=0; i<1000000; i++)
17      {
18          cout << "I'm wasting my time debugging a for loop" << endl;
19      }
20
21      double z = cos(10.0) / atan(20.0);
22
23      cout << "z has a value of: " << z << endl;
24      return 0;
25  }
26
```



Debugging

Learn how to use the debugger!

It can drastically speed up the programming process!

Which means less frustration

and more time doing the things you love!

Topic #1: Include Etiquette

A. Never include a CPP file in another .CPP or .H file.

file1.cpp

```
void someFunc()
{
    cout << "I'm cool!";
}
```

file2.cpp

```
#include "file1.cpp" // bad!

void otherFunc()
{
    cout << "So am I!";
    someFunc();
}
```

You'll get linker errors.

Only include .H files within a .CPP file.

Topic #1: Include Etiquette

B. Never put a "using namespace" command in a header file.

someHeader.h

```
#include <iostream>

using namespace std;
```

hello.cpp

```
#include "someHeader.h"

// BUT I DON'T WANT THAT
// NAMESPACE! YOU BASTARD!

int main()
{
  cout << "Hello world!";
}
```

So this is bad…

This is called "Namespace Pollution"

Why? The .H file is forcing CPP files that include it to use its namespace.

And that's just selfish.

Instead, just move the "using" commands into your C++ files.

Topic #1: Include Etiquette

alcohol.h

```
class Alcohol
{
   ...
};
```

student.h

```
#include "alcohol.h"

class Student
{
   ...
private:
   Alcohol bottles[5];
};
```

main.cpp

```
#include "student.h"

int main()
{
   Student larry;
   Alcohol vodka;

   cout << larry << " is drinking " << vodka;
}
```

C. Never assume that a header file will include some other header file for you.

In this example, main.cpp defines an Alcohol variable (vodka) but it doesn't #include"alcohol.h". That's a bad thing, even though in this example the code will compile. Why will it work in this case? Because main.cpp includes student.h, and right now, student.h includes alcohol.h. So main.cpp transitively gets alcohol.h included for it. But, a cpp file must NEVER rely upon another header file to include a header file for it. That causes problems. For example, what if the coder who wrote student.h decides to change it and remove the alcohol? Let's go to the next slide and see.

Topic #1: Include Etiquette

C. Never assume that a header file will include some other header file for you.

alcohol.h

```
class Alcohol
{
   ...
};
```

student.h

```
#include "redbull.h"

class Student
{
   ...
private:
   RedBull bottles[5];
};
```

- Utoh! Someone changed our student.h file. Now main.cpp no longer compiles! ☹
- Why? Because it doesn't include alcohol.h but it defines an Alcohol variable!
- main.cpp made the assumption that it would just get the alcohol header file included for it. But that's a bad assumption, because if someone else changes the student.h file (changing from alcohol to redbull), everything breaks.
- So the moral of the story is that if a cpp file needs to define a variable of type X (e.g., X=Alcohol), then it must include the x.h header itself, and not expect some other header to include the header implicitly.
- So how do we fix our code? Next slide!

main.cpp

```
#include "student.h"

int main()
{
   Student larry;
   Alcohol vodka;

   cout << larry << " is drinking " << vodka;
}
```

Topic #1: Include Etiquette

C. Never assume that a header file will include some other header file for you.

- To fix the code, simply #include the proper header file directly in main.cpp where you're using alcohol
- That way, your main.cpp guarantees that no matter how much student.h changes, it won't impact its ability to define an alcohol variable.
- Now your main.cpp file will compile correctly regardless of what's contained in the other header files it uses!

alcohol.h

```
class Alcohol
{
   ...
};
```

student.h

```
#include "redbull.h"

class Student
{
   ...
private:
   RedBull bottles[5];
};
```

main.cpp

```
#include "student.h"
#include "alcohol.h"
int main()
{
   Student larry;
   Alcohol vodka;

   cout << larry << " is drinking " << vodka;
}
```

Topic #2: Preprocessor Directives

C++ has several special commands which you sometimes need in your programs.

Command 1: #define

- You can use the #define command to define new constants (like PI).
- This technique was used in the original C language to define constants like PI for use in your program.
- The C preprocessor would do a search-and-replace for the constant string "PI" in your program and replace it with the associated value "3.14159", then compile the program normally.
- But #define is NOT used any more for this purpose in modern C++. Why? Because in C++ we want every variable/constant to have an explicit type (like int, double, etc.), but with #define you don't specify the data type when you define your constant.
- So now #define is only used to aid in proper compilation of programs. We'll see how in a bit.
- Oh, one more thing - you can also use #define to define a new constant without specifying a value (like FOOBAR).
- Why you ask? We'll see!

file.cpp

```
#define PI 3.14159
#define FOOBAR

void someFunc()
{
    cout << PI;
}
```

Preprocessor Directives

Command 2: #ifdef and #endif

- You can use the #ifdef command to check if a constant has been defined already…
- The compiler only compiles the code between the #ifdef and the #endif if the constant was defined above the #ifdef
- So if the constant wasn't defined, the #ifdef essentially turns into a /* and the #endif turns into a */, commenting out the whole section of code

file.cpp

```
#define FOOBAR

#ifdef FOOBAR
void someFunc()
{
    cout << PI;
}
#endif
```

C++ Compiler: Since FOOBAR was defined above, I'll compile the code until #endif.

file.cpp

```
 define FOOBAR

#ifdef FOOBAR
void someFunc()
{
    cout << PI;
}
#endif
```

C++ Compiler: Since FOOBAR was NOT defined above, I'll ignore the code until #endif as if it were commented out with /* and */

Preprocessor Directives

Command 2: #ifndef and #endif

- Just as we have #ifdef, we also have #ifndef (if NOT defined)
- You can use the #ifndef command to check if a constant has NOT been defined already…
- The compiler only compiles the code between the #ifndef and the #endif if the constant was NOT defined above.
- So if the constant was defined, the #ifndef essentially turns into a /* and the #endif turns into a */, commenting out the whole section of code

file.cpp

```
#define FOOBAR

#ifndef FOOBAR
void someFunc()
{
    cout << PI;
}
#endif
```

C++ Compiler:
Since FOOBAR was defined above, I'll ignore the code until #endif.

file.cpp

```
 define FOOBAR

#ifndef FOOBAR
void someFunc()
{
    cout << PI;
}
#endif
```

C++ Compiler:
Since FOOBAR was NOT defined above I'll compile the code until #endif.

Separate Compilation

calc.h

```
class Calculator
{
public:
   int compute();
   ...
};
```

calc.cpp

```
#include "calc.h"

int Calculator::compute()
{
   ...
}
```

student.h

```
#include "calc.h"

class Student
{
public:
   void study();
   ...
private:
   Calculator myCalc;
};
```

student.cpp

```
#include "student.h"

void Student::study()
{
 cout << myCalc.compute();
}
```

main.cpp

```
#include "student.h"
#include "calc.h"

int main()
{
   Student grace;
   Calculator hp;
   ...
   grace.study();
   hp.compute();
}
```

- When using class composition, it helps to define each class in a separate pair of .cpp/.h files. For instance, we put our Calculator class in calc.h and our Student class in student.h.
- Then you can #include them individually and use them in your program…
- Now – something interesting (and bad) happens if we #include a file A (calc.h) and file B (student.h) in the same file (main.cpp), and file B (student.h) also #includes file A (calc.h) due to class composition.
- Notice that main.cpp #includes both student.h and calc.h.
- Further notice that student.h ALSO includes calc.h.
- Hmmm!!!

Separate Compilation Problems

- So what's the problem? Well, since main.cpp includes both student.h and calc.h (#1 and #2), and student.h ALSO includes calc.h (#3), during compilation we end up with two definitions of Calculator in main.cpp.
- Why? During compilation, the compiler will "pre-process" main.cpp and generate a temporary main.cpp file like the one to the right (for final compilation). First the preprocessor processes #include #1 and this copies the contents of student.h into the temp main.cpp. While doing this, the preprocessor sees #include #3 and copies calc.h's contents to the top of the temp main.cpp file. See #5 and #6 for the Calculator definition and Student definition due to the first #include of student.h by #1.
- After including student.h, main.cpp then directly #includes calc.h (#2). This causes the contents of calc.h to be copied again to the temp main.cpp file (#7).
- The result is two definitions of the Calculator class in the same main.cpp file (#5 and #7), which is not allowed in C++. This results in a compiler error.

calc.h

```
class Calculator
{
public:
   int compute();
   ...
};
```

#1

#2

main.cpp

```
#include "student.h"
#include "calc.h"

int main()
{
   Student grace;
   Calculator hp;
   ...
   grace.study();
   hp.compute();
}
```

#3

#4

student.h

```
#include "calc.h"

class Student
{
public:
   void study();
   ...
private:
   Calculator myCalc;
};
```

#5

#6

#7

main.cpp (temp version)

```
class Calculator
{
public:
   void compute();
   ...
};
class Student
{
public:
   void study()
   ...
private:
   Calculator myCalc;
};
class Calculator
{
public:
   void compute();
   ...
};
int main()
{
   Student grace;
   Calculator hp;
   ...
   grace.study();
   hp.compute();
}
```

Fixing Separate Compilation Problems

- To fix this problem you must add "include guards" to the top and bottom of ALL header files.
- An include guard is a special check that prevents duplicate header inclusions in the same file (like main.cpp).
- If your header file is called xyz.h, then add the following lines to the VERY top of the header file:
  #ifndef XYZ_H
  #define XYZ_H
- And add the following line to the VERY end of the header file:
  #endif // XYZ_H
- See the examples below on the left for calc.h and student.h.
- Now after you #include your headers in main.cpp, this is what it will look like when the compiler generates a temp version of main.cpp that includes the header file contents (See main.cpp initial temp version)
- *Due to the include guards, when this temp version of main.cpp is compiled, the compiler will get to line #1 and see that CALC_H is not defined, so it will #define CALC_H on line #2 and then include the first definition of Calculator.*
- *When the compiler reaches the second #ifndef CALC_H on line #3, the CALC_H constant will have been defined (on line #2), and so C++ will skip over the second definition of calculator, ignoring the code through line #4. The final code will look like the one on the right. So there's only one definition of the Calculator class in main.cpp and C++ is happy.*

calc.h

```
#ifndef CALC_H
#define CALC_H

class Calculator
{
public:
  void compute();
  ...
};
#endif // for CALC_H
```

student.h

```
#ifndef STUDENT_H
#define STUDENT_H

#include "calc.h"

class Student
{
public:
  void study()
  ...
private:
  Calculator myCalc;
};
#endif // for STUDENT_H
```

main.cpp (initial temp version)

```
#ifndef STUDENT_H
#define STUDENT_H

#ifndef CALC_H          ← #1
#define CALC_H          ← #2
class Calculator
{
public:
  void compute();
};
#endif // for CALC_H
class Student
{
public:
  void study()
private:
  Calculator myCalc;
};
#endif // for STUDENT_H

#ifndef CALC_H          ← #3
#define CALC_H
class Calculator
{
public:
  void compute();
};
#endif // for CALC_H    ← #4
int main()
{
  Student grace;
  Calculator hp;
  ...
  grace.study();
  hp.compute();
}
```

main.cpp (final temp version)

```
#ifndef STUDENT_H
#define STUDENT_H

#ifndef CALC_H
#define CALC_H
class Calculator
{
public:
  void compute();
};
#endif // for CALC_H
class Student
{
public:
  void study()
private:
  Calculator myCalc;
};
#endif // for STUDENT_H

#ifndef CALC_H
#define CALC_H
class Calculator
{
public:
  void compute();
};
#endif // for CALC_H
int main()
{
  Student grace;
  Calculator hp;
  ...
  grace.study();
  hp.compute();
}
```

alcohol.h

```
class Alcohol
{
public:
    void drink() { cout << "glug!"; }
};
```

student.h

```
#include "alcohol.h"

class Student
{
public:
  void beIrresponsible();
  ...
private:
  Alcohol *myBottle;
};
```

I use the Alcohol class (which is defined in alcohol.h) to define a member variable

So I need to include alcohol.h, right?

Last Topic: Knowing *WHEN* to Include .H Files

You might think that any time you refer to a class, you should include its .h file first… Right?

Well, not so fast!

A.h

```
class FirstClass
{
public:
    void someFunc() { ... }
};
```

B.h

```
#include "A.h"

class SecondClass
{
public:
  void otherFunc()
  {
    FirstClass y;
    FirstClass b[10];
    y.someFunc();
    return(y);
  }

private:
  FirstClass x;
  FirstClass a[10];
};
```

Last Topic: Knowing *WHEN* to Include .H Files

- Here are the rules: You must include the header file (containing the full definition of a class) any time:
  1. You define a regular variable of that class's type, OR
  2. You use the variable in any way (call a method on it, return it, etc).
- Why? Because C++ needs to know the class's details in order to define actual variables with it or to let you call methods from it!
- On the other hand…

Last Topic: Knowing *WHEN* to Include .H Files

A.h

```
class FirstClass
{
public:
    void someFunc() { ... }
};
```

B.h

```
class FirstClass;  // #1

class SecondClass
{
public:
   void goober(FirstClass p1);
   FirstClass hoober(int a);
   void joober(FirstClass &p1);
   void koober(FirstClass *p1);

   void loober()
   {
      FirstClass *ptr;
   }

private:
  FirstClass *ptr1, *z[10];
};
```

- If you do NONE of the previous items, but you…
  1. Use the class to define a parameter to a function, OR
  2. Use the class as the return type for a func, OR
  3. Use the class to define a pointer or reference variable
- Then you DON'T need to include the class's .H file.
- (You may often do so, but you don't need to)
- Instead, all you need to do is give C++ a hint that your class exists (and is defined elsewhere).
- Look at line #1 to see how you do that.
- **In your .h header file**, you simply write the word "class" followed by the class name you want C++ to know about INSTEAD of using the #include.
- This line tells C++ that your class exists, but doesn't actually define all the gory details.
- Since none of the code in file B.h actually uses the "guts" of the class (as the code did on the previous slide), this is all C++ needs to know to define the header file!
- **Then, in your .cpp file**, you do #include the required header file at the top of the file (the one that defines FirstClass) normally.
- This allows your .cpp file to have the full definition of the class so it can call its functions and access its data members,.

A.h

```
class FirstClass
{
   … // thousands of lines
   … // thousands of lines
};
```

B.h

```
#include "A.h" // really slow!

class SecondClass
{
public:









private:

};
```

Last Topic: Knowing *WHEN* to Include .H Files

Wow – so confusing!

Why not just always use #include to avoid the confusion?

There are two reasons:

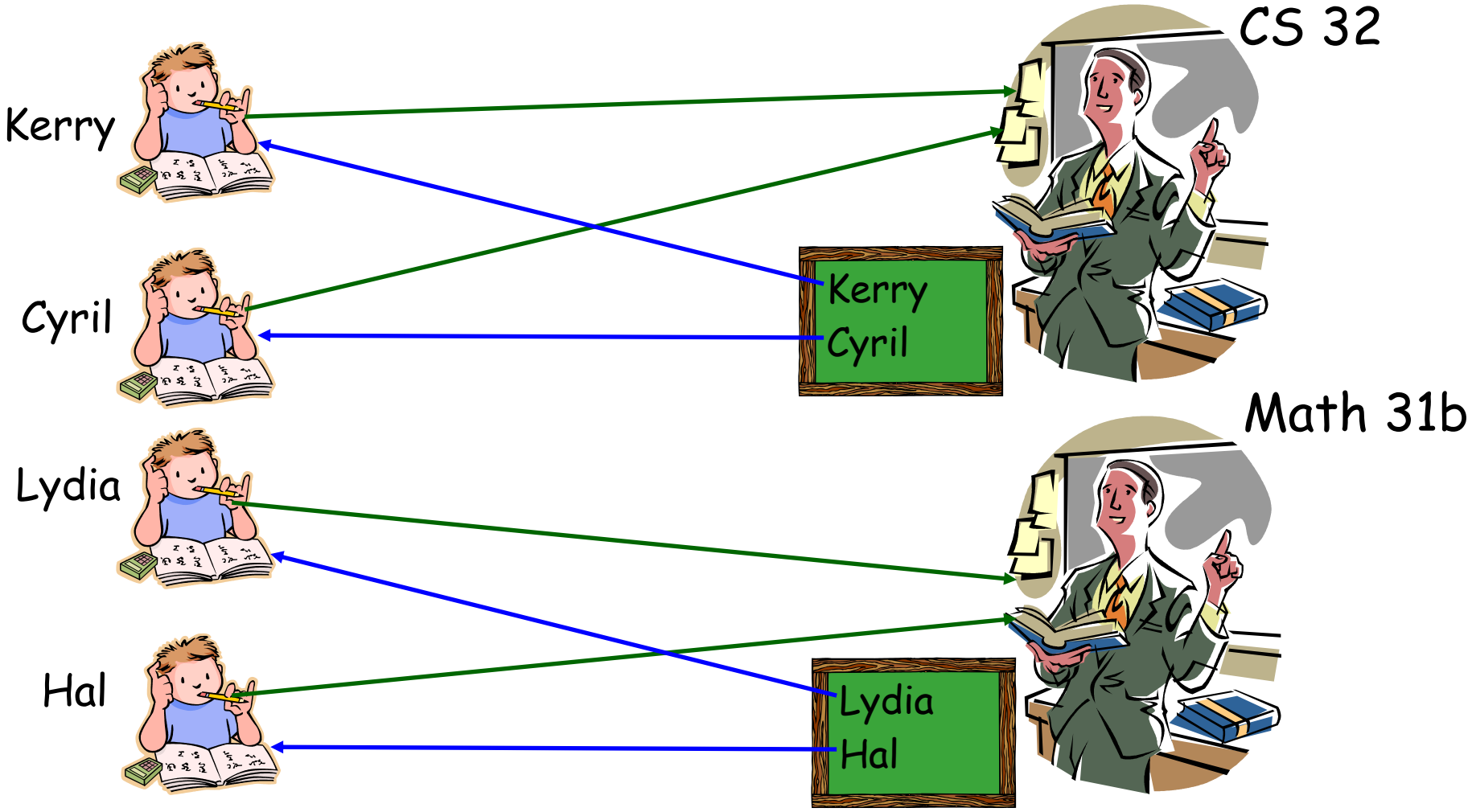
1. If a .h file is large (thousands of lines), #including it when you don't strictly need to slows down your compilation. Especially if you're compiling thousands of files that include your really long .H file.

2. If two classes refer to each other, and both classes try to #include the other's .H file, you'll get a compile error. Let's see that case.s

Students and Classrooms

Every student knows what class he/she is in…

Every class has a roster of its students…



These type of cyclical relationships cause #include problems!

Last Topic: Self-referential Classes

- Our ClassRoom class refers to the Student class so it includes Student.h
- And our Student class refers to the Classroom class, so it includes Classroom.h
- Do you see the problem?
- Student.h tries to include Classroom.h which tries to include Student.h which tries to include Classroom.h and on and on.
- This will cause the compiler to die! The include guards we learned won't solve the problem (try for yourself to see why)

Student.h

```
#include "ClassRoom.h"

class Student
{
public:
  ...

private:
   ClassRoom *m_myRoom;
};
```

ClassRoom.h

```
#include "Student.h"

class ClassRoom
{
public:
  ...

private:
   Student m_studs[100];
};
```

Student.cpp

```
#include "Student.h"

void Student::printMyClassRoom()
{
 cout <<"I'm in Boelter #" <<
    m_myRoom->getRmNum();
}
```

ClassRoom.cpp

```
#include "ClassRoom.h"

void ClassRoom::printRoster()
{
 for (int i=0;i<100;i++)
  cout << m_studs[i].getName();
}
```

So how do we solve this cyclical nightmare?

Step #1:

Look at the two class definitions in your .h files. At least one of them should NOT need the full #include.

Question:

Which of our two classes doesn't need the full #include? Why?

Student.h

```
#include "ClassRoom.h"

class Student
{
public:
    ...

private:
    ClassRoom *m_myRoom;
};
```

This class JUST defines a pointer to a ClassRoom. It does NOT hold a full ClassRoom variable and therefore doesn't require the full class definition here!!!

ClassRoom.h

```
#include "Student.h"

class ClassRoom
{
public:
    ...

private:
    Student m_studs[100];
};
```

This class defines actual full Student variables... It therefore requires the full class definition to work!

So how do we solve this cyclical nightmare?

Step #2:

Take the class that does NOT require the full #include
(forget the other one) and update its header file:

Replace the #include "XXXX.h" statement
with the following: class YYY;
Where YYY is the other class name (e.g., ClassRoom).

This line tells the compiler:
"Hey Compiler, there's another class called 'ClassRoom' but we're not going to tell you exactly what it looks like just yet."

Student.h

```
class ClassRoom;

class Student
{
public:
    ...

private:
    ClassRoom *m_myRoom;
};
```

ClassRoom.h

```
#include "Student.h"

class ClassRoom
{
public:
    ...

private:
    Student m_studs[100];
};
```

So how do we solve this cyclical nightmare?

Student.h

```
class ClassRoom;

class Student
{
public:
   ...

private:
   ClassRoom *m_myRoom;
};
```

Student.cpp

```
#include "Student.h"
#include "ClassRoom.h"
void Student::printMyClassRoom()
{
 cout <<"I'm in Boelter #" <<
     m_myRoom->getRmNum();
}
```

Step #3:

Almost done. Now update the CPP file for this class by #including the other header file normally.

This line tells the compiler:

"Hey Compiler, since my CPP file is going to actually use ClassRoom's functionality, I'll now tell you all the details about the class."

The compiler is happy as long as you #include the class definition before you actually use the class in your functions…

So how do we solve this cyclical nightmare?

Step #4:

Finally, make sure that your class doesn't have any other member functions that violate our #include vs class rules…

- If you have defined the full {body} of one or more functions DIRECTLY in your class definition in your .h file AND they use your other class in a significant way, then you must MOVE them to the CPP file.
- Consider the definition of the beObnoxious() function in Student.h (#1). The function uses the Classroom variable m_myRoom by calling the getRmNum() method. That won't work anymore since Student.h doesn't have a full definition of the Classroom class anymore.
- So instead, just define the function prototype (#2) in your .h file, and move the full definition of your function to your .cpp file (#3), after you've included the proper header file.

Student.h

```
class ClassRoom;

class Student
{
public:
  …
 void beObnoxious() /* #1 */ {
   cout << m_myRoom->getRmNum() << " sucks!"; }
private:
  ClassRoom *m_myRoom;
};
```

Student.h

```
class ClassRoom;

class Student
{
public:
  …
 void beObnoxious(); // #2: just define prototype

private:
  ClassRoom *m_myRoom;
};
```

Student.cpp

```
#include "Student.h"
#include "ClassRoom.h"




... // code
```

Student.cpp

```
#include "Student.h"
#include "ClassRoom.h"  // now we have the full classroom def!

// write the full function { body} here!
void Student::beObnoxious() {    // #3
  cout << m_myRoom->getRmNum() << " sucks!"; }

... // code
```

So how do we solve this cyclical nightmare?

Now let's look at both of our classes… Notice, we no longer have a cyclical reference! Woohoo!

Student.h

```
class ClassRoom;

class Student
{
public:
   ...

private:
  ClassRoom *m_myRoom;
};
```

ClassRoom.h

```
#include "Student.h"

class ClassRoom
{
public:
  ...

private:
    Student m_studs[100];
};
```

Student.cpp

```
#include "Student.h"
#include "ClassRoom.h"
void Student::printMyClassRoom()
{
 cout <<"I'm in Boelter #" <<
     m_myRoom->getRmNum();
}
```

ClassRoom.cpp

```
#include "ClassRoom.h"

void ClassRoom::printRoster()
{
 for (int i=0;i<100;i++)
  cout << m_studs[i].getName();
}
```

Default Arguments!

```
void poop(string name, int numberOfWipes)
{
  cout << name << " just pooped.\n";
  cout << "They wiped " << numberOfWipes <<
    " times.\n";
}

...

int main()
{
    poop("Phyllis",2);
    poop("Astro",2);
    poop("Sylvia",2);
    poop("Carey",3);
    poop("David",0);
    poop("Sergey",2);
    poop("Larry",2);
    poop("Devan",2);
}
```

So what do you notice about the following code?

(other than David really needs to wipe)

Right! Most of the time, people wipe exactly twice!

Wouldn't it be nice if we could simplify our program by taking this into account?

Default Arguments!

```
void poop(string name, int numberOfWipes = 2)
{
  cout << name << " just pooped.\n";
  cout << "They wiped " << numberOfWipes <<
    " times.\n";
}

...

int main()
{
    poop("Phyllis"  ); // defaults to passing in 2
    poop("Astro"    ); // defaults to passing in 2
    poop("Sylvia",7);
    poop("Carey",3);
    poop("David",0);
    poop("Sergey"   ); // defaults to passing in 2
    poop("Larry"    ); // defaults to passing in 2
    poop("Devan"    ); // defaults to passing in 2
}
```

Note: We still need to pass in the value any time we don't want the default!

We can!
Let's see how!

C++ lets you specify a default value for a parameter… like this…

Now, when you call the function, you can leave out the parameter if you just want the default…

and C++ will automagically pass in that default value!

Default Arguments!

```
void fart(int length = 10, int volume = 50)
{
   cout << "I just farted for " << length
     << " seconds, and it was"
     << volume << " decibels loud.\n"
}

int main()
{
   fart(20,5);     // long but not so loud
   fart(5);        // short violent burst!
   fart();         // medium and pretty loud
   fart(,30);      // NOT ALLOWED
}
```

You can have more than one default parameter if you like, and C++ will figure out what to do!

But you can't do something like this!

Why not? Good question – ask the C++ language designers. ☺

Default Arguments: One other thing

If the j^{th} parameter has a default value, then all of the following parameters (j+1 to n) must also have default values.

```
// INVALID! Our 1st param is default, but 2nd and 3rd aren't!
bool burp(int length = 5, int loudness, int pitch)
```

```
// INVALID! Our first 2 params are default, but 3rd isn't!
bool burp(int length = 5, int loudness = 12, int pitch)
```

```
// INVALID! Our 1st param is default, but 2nd isn't!
bool burp(int length = 5, int loudness, int pitch = 60)
{
   ...
}
```

```
// PERFECT! All parameters are default
bool burp(int length = 5, int loudness = 5, int pitch = 60)
```

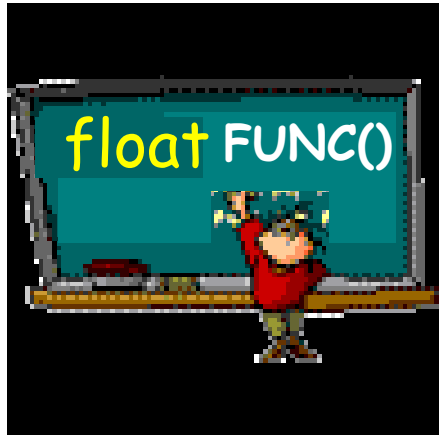
```
// PERFECT! All default parameters come after non-defaults
bool burp(int length, int loudness = 5, int pitch = 60)
{
   ...
}
```

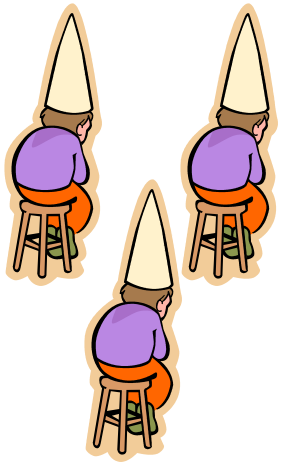
Class Challenge

RULES

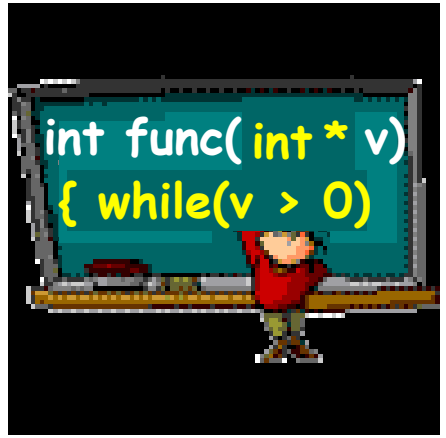
- The class will split into left and right teams
- One student from each team will come up to the board
- Each student can either
  - write one new line of code to solve the problem OR
  - fix a single error in the code their teammates have already written
- Then the next two people come up, etc.
- The team that completes their program first wins!

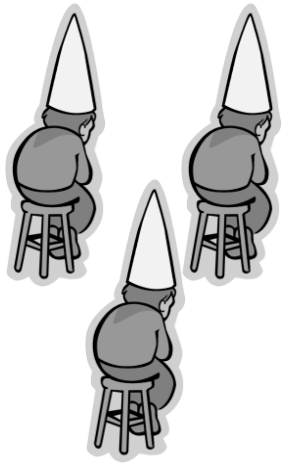
Team #1





Team #2





Challenge #1

Write a class called Quadratic which represents a second-order quadratic equation, e.g.: $4x^2+3x+5$

When you construct a Quadratic object, you pass in the three coefficients (e.g., 4, 3, and 5) for the equation.

The class also has an evaluate method which allows you pass in a value for x. The function will return the value of f(x).

The class has a destructor which prints out "goodbye".

Challenge #2

Write a class called MathNerd which represents a math nerd. Every MathNerd has his own special quadratic equation!

When you construct a MathNerd, he always wants you to specify the first two coefficients (for the x^2 and x) for his equation.

The MathNerd has a getMyValue function which accepts a value for x and should return f(x) for the nerd's QE.