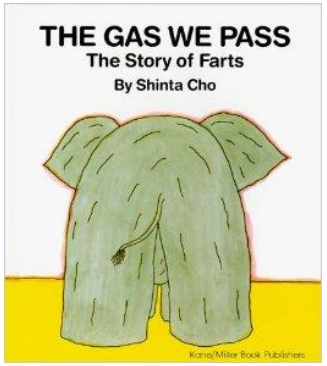
Lecture #2

(aka the Fart Lecture)

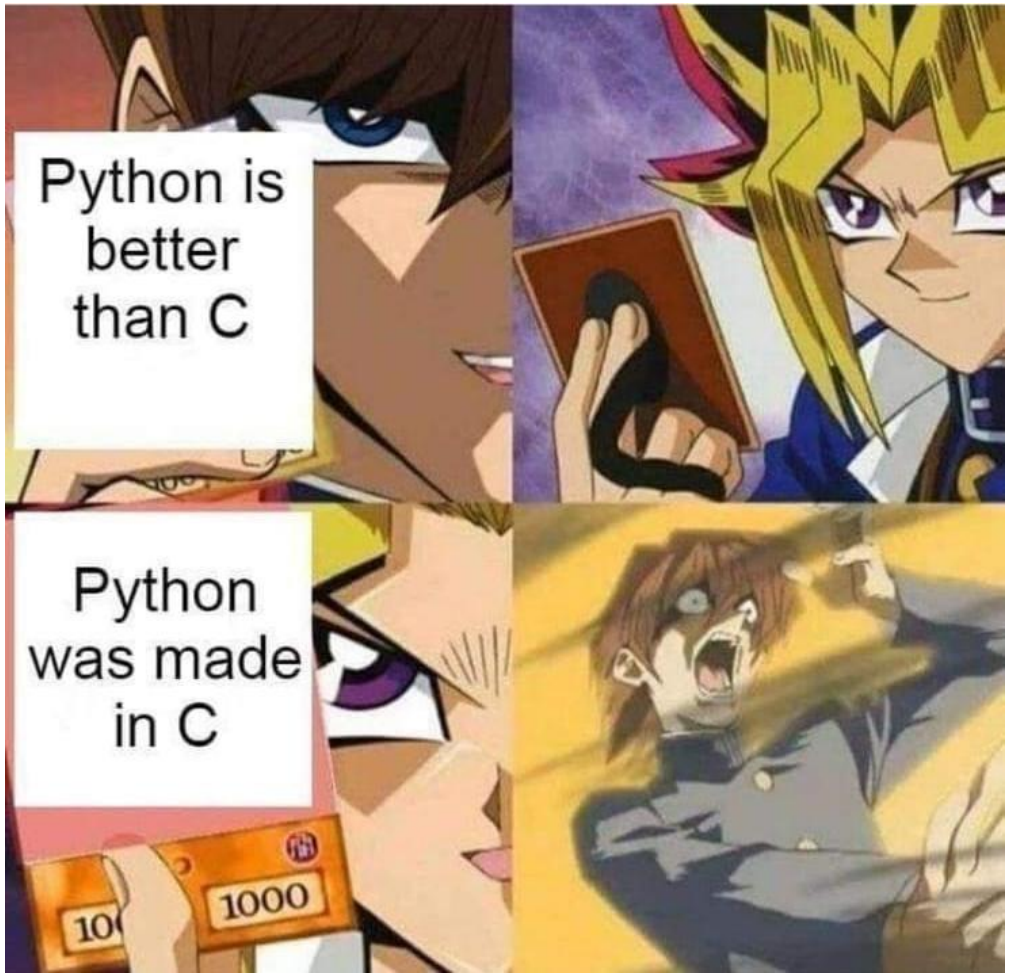


- Part 1: Basic C++ Concepts
  - Constructors
  - Destructors
  - Class Composition
  - Composition with Initializer Lists
- Part 2: On-your-own Study Topics
  - Learning how to use the Visual C++/Xcode debuggers
  - A few final topics you need to know to do Project #1

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)!

Classrooms for each section will be posted on the class website!

Meme, anyone?



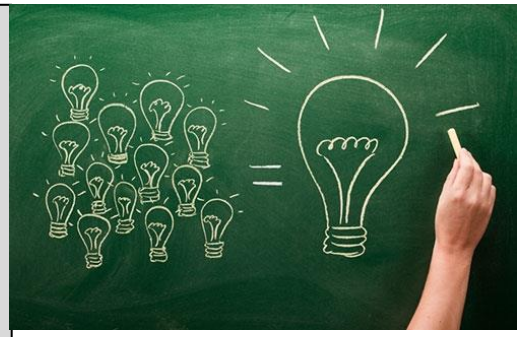
Constructors and Destructors

What's the big picture?

Constructors

A constructor function is used to reset an object's member variables when the object is first created; otherwise they'd be random!

```
class SomeClass {
public:
   void blah() { cout << v; }  // random value!
private:
   int v;  // uninitialized if no constructor!
};
```



Destructors

An object will often reserve memory slots from the operating system while it runs.

A destructor function guarantees that reserved memory is freed when an object goes away.

If you don't free this memory, your program will eventually run out of memory and crash.

Uses:

Constructors and destructors are required for virtually every correct C++ class.

Constructor Basics

```
class Gassy
{
public:
  Gassy(int age, bool ateBeans)
  {
     m_age      = age;
     m_ateBeans = ateBeans;
  }

  int getFartsPerHr()
  {
     if(m_ateBeans == true)
       return(100);
     return(3 * m_age);
  }

private:
  int m_age;
  bool m_ateBeans;
};
```

Has 2 Params!

Here's our Gassy class….
It represents gassy people.

- Our Gassy class has a constructor that takes two parameters to initialize it.
- If a constructor has parameters, then you must pass values in to them that match the types of the parameters!
- Notice down below, that when we construct betty we pass in two values, so it works just fine
- But when we construct alan, we don't pass any parameters in, so this would fail.

Requires two args!

```
int main()
{
  Gassy betty(18,true);

  Gassy alan;  // error!

}
```

```
class Gassy
{
public:
  Gassy(int age, bool ateBeans)
  {
    m_age = age;
    m_ateBeans = ateBeans;
  }
  Gassy()
  {
    m_age = 0;
    m_ateBeans = false;
  }

  int getFartsPerHr()
  {
    if(m_ateBeans == true)
      return(100);
    return(3 * m_age);
  }
private:
  int m_age;
  bool m_ateBeans;
};
```

Constructor Basics

- Your class can have many different constructors…
- The only requirement is that each one has different parameters and/or types.
- This is called constructor overloading.
- So now, our definition below of the alan variable will work, and will result in a call to the second constructor.
- Notice that we don't need (and in fact, we are not allowed to have) parentheses after the definition of the alan variable.

```
int main()
{
  Gassy betty(18,true);

  Gassy alan;  // OK!!!

}
```

Constructor Basics

```
class Gassy
{
public:
 Gassy() // generated by compiler
 {
   // I don't init primitive member
   // variables… So be careful!!!
 }
 int getFartsPerHr()
 {
   if(m_ateBeans == true)
     return(100);
   return(3 * m_age);
 }

private:
  int m_age;
  bool m_ateBeans;
};
```

These variables will have random values!

- If you don't define any constructors at all then C++ generates an implicit, default constructor for you.
- But this default constructor won't initialize your object's primitive member variables!
- Primitive variables include native C++ types like int, float, double, bool, etc.
- Non-primitives include string, Gassy, etc.
- So the call to getFartsPerHr() below will result in random values being printed.

```
int main()
{
 Gassy    carey;

 cout <<
   carey.getFartsPerHr(); //??
}
```

When Constructors are Called

```
int main()
{
    Gassy  carey(45,false), bill;

    Gassy  arr[52];

    Gassy  *ptr = new Gassy(1,false);

    for (int i=0; i < 10; ++i) {
        Gassy temp;
        ...
    }

    Gassy  *justAPtr;
}
```

A constructor is called any time you create a new variable of a class.

A constructor is called N times (once for each array element) when you create an array of size N.

A constructor is called when you use new to dynamically allocate a new variable.

If a variable is declared in a loop, it is newly constructed during EVERY iteration!

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

And now it's time for…

Figure out what the obfuscated C program does!

Destructors

```
class Gassy
{
public:

  ~Gassy()
  {
     // your destructor
     // code goes here
  }

private:
     …
};
```

- Just as every class has a constructor, every class also has a destructor function (it may have *only one* of these).
- The job of the destructor is to de-initialize or destruct a class variable when it goes away.
- Destructors must NOT have any parameters.
- To define a destructor function, place a tilde ~ character in front of the name of the class.
- It looks just like a constructor function except for the tilde ~ which identifies it as a destructor.
- Destructors must NOT return a value either. They *can* use the return; command, they just can't return a value, e.g. return 7;

Destructors

```
class Gassy
{
public:
  Gassy()
  {
     m_age = 0;
     m_ateBeans = false;
  }

  int getNerdScore()
  {
    if(m_ateBeans == true)
      return(100);
    return(3 * m_age);
  }
  ~Gassy() // generated by compiler
  {
  }
private:
   int m_age;
   bool m_ateBeans;
};
```

- If you don't define your own destructor for a class…
- Then C++ will define an implicit one for you…
- This ensures that objects of your class properly go away when they go out of scope.
- We'll see exactly why in a bit.

Why Do We Need Destructors?

```
class SpotifyPlayer
{
public:
    SpotifyPlayer()
    {
        // Reserve 100MB of disk space to temporarily
        // cache downloaded MP3 music files.
        reserveSpaceOnDisk(100000000);
    }


    void playSong(string songURL)
    {
        // First download song to reserved space on disk.
        downloadMP3ToReservedSpace(songURL);
        // Then play the song that was saved to our disk.
        playMP3InReservedSpace();
    }



};
```

- Can anyone see what the problem is?
- Every time we construct a new SpotifyPlayer variable, it reserves 100MB on disk.
- But our SpotifyPlayer class never deletes the reserved space!
- Now, as it turns out, if you define a class variable inside a loop, C++ will create a new variable from scratch EVERY TIME THE LOOP RUNS!
- So every time the loop runs, it creates a whole new SpotifyPlayer variable and reserves 100MB
- If the loop runs 10 times, you'll end up reserving 1GB of space!
- Eventually your hard drive will run out of disk space.

```
int main()
{
    // Obsessively play k-pop music!
    for (int i = 0; i < 100; ++i)
    {
        SpotifyPlayer a;  // ← a's c'tor runs every loop iteration!
        a.playSong("http://spotify.com/dad.mp3");
    }
}
```

Why Do We Need Destructors?

```
class SpotifyPlayer
{
public:
   SpotifyPlayer()
   {
      // Reserve 100MB of disk space to
temporarily
      // cache downloaded MP3 music files.
      reserveSpaceOnDisk(100000000);
   }
  ~SpotifyPlayer()
   {
      // Free the reserved space on disk.
      freeReservedSpace();
   }

   void playSong(string songURI
   {
      // First download song t
      downloadMP3ToReserved
      // Then play the song t
      playMP3InReservedSpace
   }

};
```

- Once we add the destructor to SpotifyPlayer we've fixed the problem!
- Why? If you define a class variable inside a loop (like variable a below), C++ will run the variable's destructor EVERY TIME THE LOOP reaches the closing bracer of the loop!
- So every time the loop runs, it creates a whole new SpotifyPlayer variable and reserves 100MB
- And every time the loop runs, when it hits the closing bracer } of the loop, C++ runs the destructor of the variable.
- If the loop runs 10 times, C++ will reserve 100MB of space where the variable is defined, and it will free the reserved 100MB where each time it reaches the } in the loop.
- So +100MB in, -100MB out, and by the end of each loop iteration, you have 0 disk space reserved.

```
int main()
{
  // Obsessively play k-pop music!
  for (int i = 0; i < 100; ++i)
  {
    SpotifyPlayer a;  // ← a's c'tor runs every iteration
    a.playSong("http://spotify.com/dad.mp3");
  } // ← a's destructor runs every loop iteration!
}
```

When must you have a destructor?

Any time a class allocates a system resource…

Reserves memory using the *new* command

Opens a disk file

Connects to another computer over the network

Your class must have a destructor that…

Frees the allocated memory with the *delete* command

Closes the disk file

Disconnects from the other computer

Don't forget or you'll FAIL!

Destructors

So when is a destructor called anyway?

```
void Dingleberry()
{
    SomeClass a;
```

- Local objects defined in a function are destructed when they "go out of scope."
- Variables defined in a function (like variable a) go out of scope when the function exits.
- Variables defined in a { block } are destructed when the block finishes. A block is one or more lines of code surrounded by { }. So variable b will be constructed once every time the loop runs, and destructed once every time the loop runs (when the loop reaches the }).
- Dynamically allocated variables (e.g., allocated using the new command) have their destructor run ONLY when delete is called, *before* the memory is actually freed by the OS.

```
    for (int j=0;j<10;j++)
    {
      SomeClass b;
      // do something
    }
```

← b's destructor is called each time we hit the end of the block

```
    MP3Player *m = new MP3Player;
    m->getSong("www.music.com/x.wav");
    m->playSong();
    delete m;
```

← m's destructor is called here

(if you don't use delete, m's d'tor is ***never*** called – ***not even when the program ends!***)

```
}
```

← a's destructor is called here

Destructors

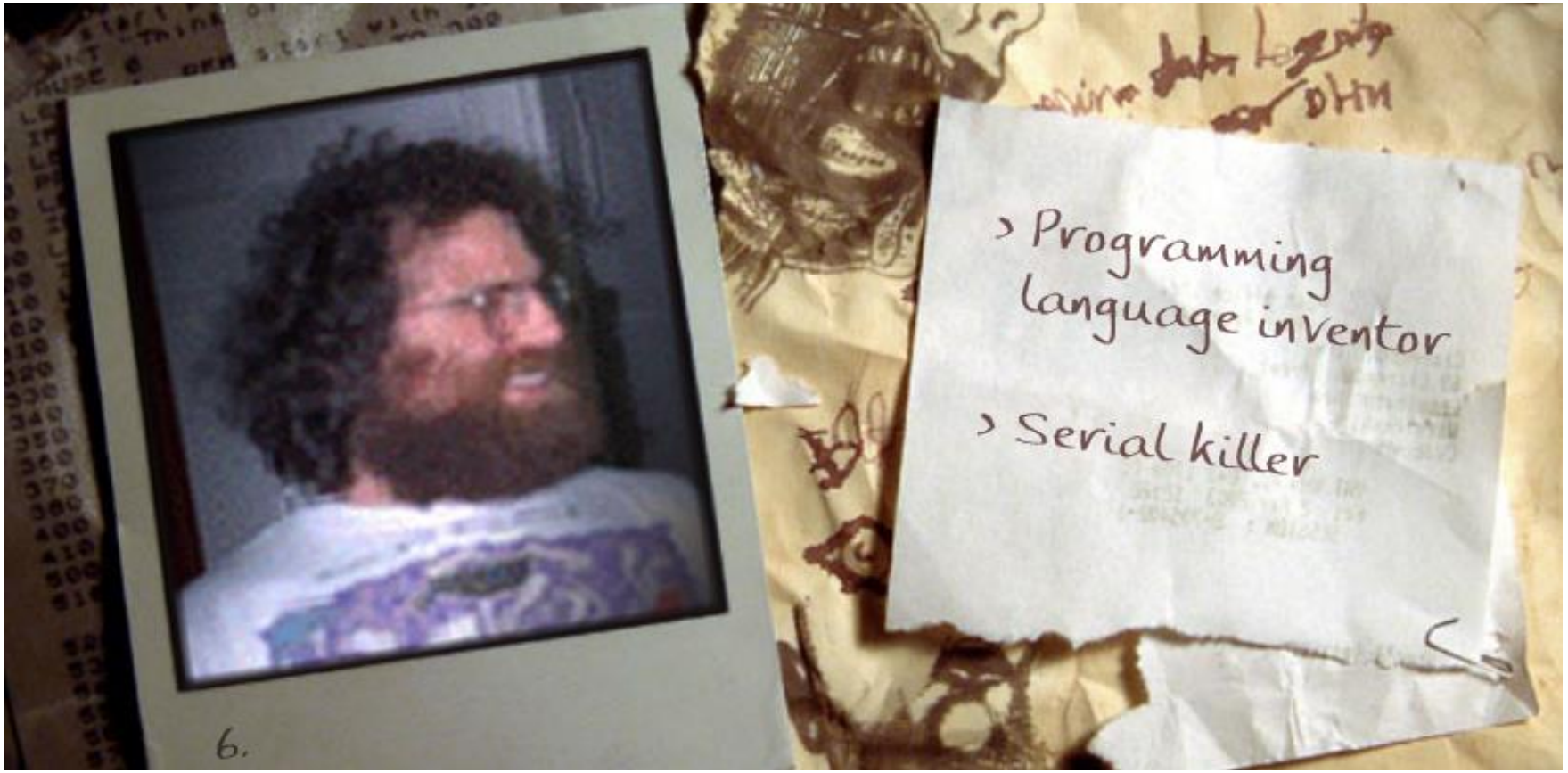
So when is a destructor called anyway?

```
void Dingleberry()
{
    SomeClass a;
    SomeClass x[42];

  for (int j=0;j<10;j++)
  {
    SomeClass b;
    // do something
  }


  MP3Player *m = new MP3Player;
  m->getSong("www.music.com/x.wav");
  m->playSong();
  delete m;


}
```

Finally, when you define an array of N items, the destructor is called N times when the array goes away…

The destructor is called 42 times for x here!

Can you guess?



See if you can guess who uses a keyboard and who uses a chainsaw!

Class Composition

What's the big picture?

Class composition is when a class contains one or more member variables that are also classes.


```
class Player {
  ...
};

class Enemy {
  ...
};

class VideoGame {
  ...
private:
  Player user_hero;
  Enemy enemies[10];
};
```

Class Composition

The important thing to watch out for in the next set of slides is the process/syntax C++ uses to initialize these member variables.

Uses:

Class composition is required in most C++ classes – you build classes using other classes!

Class Composition

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

class Brain {
public:
    Brain()       { myIQ = 100; }
    void think() { myIQ += 10; }
    ...
};

class HungryNerd {
public:
   HungryNerd()
   {
      myBelly.eat();
      myBrain.think();
   }
private:
   Stomach myBelly;
   Brain      myBrain;
};
```

Then C++ runs the outer class's constructor!

C++ constructs members first, from top to bottom.

So how does construction work when we use class composition? Let's look at three classes…

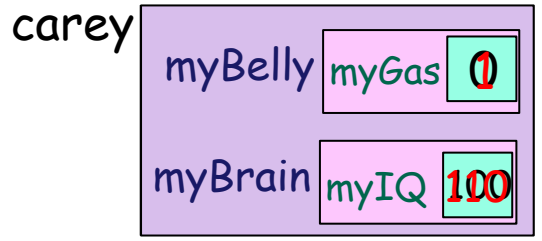
- C++ always constructs member variables first, in the order they're defined in the class (so myBelly would be constructed first, then myBrain would be constructed second).
- Finally, C++ runs the constructor of the outer variable (the nerd) last after its member variables have been initialized/constructed.
- This makes sense. Since our outer class constructor (HungryNerd) uses its member variables (myBelly.eat()), those member variables must have already been constructed or they couldn't be used!

```
class Stomach {
public:
   Stomach() { myGas = 0; }
   void eat()  { myGas ++; }
};

class Brain {
public:
    Brain()       { myIQ = 100; }
    void think() { myIQ += 10; }
};

class HungryNerd {
public:
   HungryNerd()
      Call myBelly 's default constructor
      Call myBrain's default constructor
   {
      myBelly.eat();
      myBrain.think();
   }
private:
   Stomach myBelly;
   Brain      myBrain;
};
```

This code is automagically added by the compiler!

Member objects

- How does this actually work?
- When an outer object like HungryNerd contains member objects (like myBelly and myBrain), C++ secretly adds code to the very top of the outer class's constructor to FIRST call the default constructors of all the member objects:
  - This code is added by the compiler to your final program (you won't see it in your code)
- A default constructor is one that doesn't require any parameters (like Stomach's or Brain's).
- AFTER C++ finishes constructing the member variables, C++ then runs the body of the outer object's (HungryNerd's) constructor, which can use the member variables (since they've been initialized).

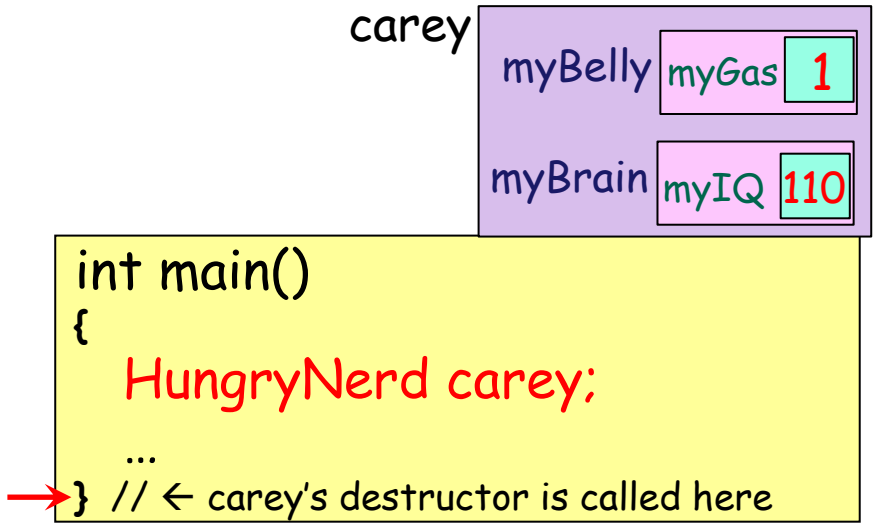



```
int main()
{
   HungryNerd carey;
   ...
}
```

```
class Stomach {
public:
   Stomach() { myGas = 0; }
   void eat()  { myGas ++; }
   ~Stomach() { cout << "Fart!\n"; }
};

class Brain {
public:
    Brain()       { myIQ = 100; }
    void think() { myIQ += 10; }
    ~Brain() { cout << "Argh!\n"; }
};
class HungryNerd {
public:
   ~HungryNerd()
   {
      myBelly.eat();      // last meal
      myBrain.think();  // last thought
   }
private:
   Stomach myBelly;
   Brain      myBrain;
};
```

So how does destruction work when we use class composition?

- At the time when our object (carey) goes out of scope, C++ automatically calls its destructor.
- C++ always runs the main/outer object's destructor first. It's member variables are still valid when the outer destructor runs. That way, the outer destructor (~HungryNerd()) has access to completely valid member variables (myBelly and myBrain) while it runs.
- After the outer class's destructor runs (and has its last meal and thinks its last thought, using valid myBelly and myBrain member variables), then C++ runs the destructors of the member **variables in the reverse order they were constructed**.
- So first ~HungryNerd() will run, then ~myBrain() will run, and finally ~myBelly() will run.



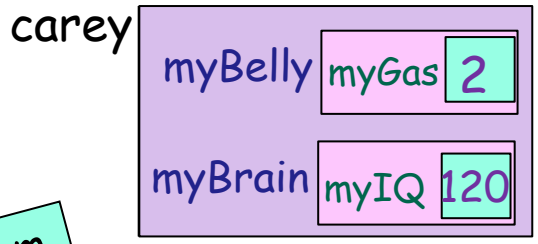
```
class Stomach {
public:
   Stomach() { myGas = 0; }
   void eat()  { myGas ++; }
   ~Stomach() { cout << "Fart!\n"; }
};

class Brain {
public:
    Brain()       { myIQ = 100; }
    void think() { myIQ += 10; }
    ~Brain() { cout << "Argh!\n"; }
};

class HungryNerd {
public:
   ~HungryNerd()
   {
      myBelly.eat();     // last meal
      myBrain.think();  // last thought
   }
    Call myBrain 's destructor
    Call myBelly 's destructor
private:
   Stomach myBelly;
   Brain      myBrain;
};
```

- How does this actually work under the hood?
- C++ secretly adds code AFTER the END of the outer destructor to call the member objects' destructors (in reverse order of their definition in the class).
- The outer destructor runs first – it can still use all of its member variables!
- Then C++ calls the d'tors of the inner objects (first destructing myBrain with ~Brain() and then destruting myBelly with ~Belly())
- And of course, if an embedded object (like myBelly) has its own embedded object(s), this process is repeated.

The member variables are still valid and can be used!

C++ "secretly" destructs them after your destructor runs.

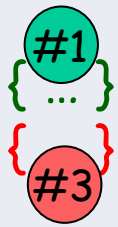
Argh!
Fart!




```
int main()
{
   HungryNerd carey;
   ...
}
```

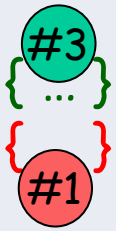
Multi-level Composition Construction/Destruction

```
class Food
{
public:
    Food()
    ~Food()
};
```




```
class Stomach
{
public:
    Stomach()
    ~Stomach()
private:
    Food    myFood;
};
```


```
class Nerd
{
public:
    Nerd()
    ~Nerd()
private:
    Stomach
myStomach;
};
```

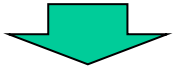


- Here's the order the c'tors will run for three levels of classes:
  - Before Nerd() can run, C++ needs to construct its myStomach variable so the Nerd() constructor can use the member variable! So it transfers control to Stomach() to run first.
  - But before Stomach() can run, C++ needs to construct Stomach's myFood variable, since the constructor may need this variable to properly run!
  - So Stomach() transfers control to Food().
  - The Food() constructor is able to run immediately, since the food class has no embedded member objects. So Food() is technically the first constructor to run when we create a new Nerd (Green #1).
  - Once Food() runs to initialize myFood, it returns to Stomach() which may now run second (Green #2), since now it's myFood member variable has been constructed.
  - Finally, Stomach() finishes and returns, so now our Nerd() constructor can run (and use its myStomach variable normally) to construct david_s (Green #3).
- Here's the order the d'tors will run for three levels of classes:
  - First ~Nerd() runs (Red #1). It needs to run before myStomach's destructor because it may need to use myStomach, and if myStomach was destructed before ~Nerd() runs, that would be impossible.
  - Once ~Nerd() has finished, C++ secretly calls the myStomach destructor ~Stomach().
  - ~Stomach() runs completely (Red #2) destructing myStomach, and once it's done, C++ secretly calls the myFood destructor ~Food().
  - ~Food() destructs myFood last (Red #3) and finally the entire Nerd and all of its contents have been destroyed.
  - C++ Can now free the memory associated with the david_s variable and its contents back to the operating system.

```
int main()
{
→Nerd      david_s;
   ...
→}
```

Back to Auto-generated Constructors and Destructors

```
class HungryNerd
{
public:
 // no user-defined c'tor or d'tor!

 void celebrate()
 {
    myBelly.eat();  // mmmm
 }
private:
   Belly myBelly;
   Brain myBrain;
};
```



```
class HungryNerd
{
public:
 HungryNerd() // implicitly added by compiler
 {
   // empty
 }

 ~HungryNerd() // implicitly added by compiler
 {
   // empty
 }

 void celebrate()
 {
    myBelly.eat();  // mmmm
 }
private:
   Belly myBelly;
   Brain myBrain;
};
```

```
class HungryNerd
{
public:
 HungryNerd() // implicitly added by compiler
   Call myBelly's default constructor
   Call myBrain's default constructor
 {
   // empty
 }
 ~HungryNerd() // implicitly added by compiler
 {
   // empty
 }
   Call myBrain's destructor
   Call myBelly's destructor
 void celebrate()
 {
    myBelly.eat();  // mmmm
 }
private:
   Belly myBelly;
   Brain myBrain;
};
```

- In the class on the top-left, we didn't define any constructors or destructors for our HungryNerd class.
- As we learned, if you leave out a constructor and/or destructor, C++ auto-generates an empty constructor and/or destructor for your class and injects them into your compiled program, as shown on the middle-left.
- But if those auto-added c'tors/d'tors have empty { bodies} why would C++ add them?
- Well, as we learned, if a class uses composition and has member objects inside of it (like myBelly or myBrain), then it also needs to construct/ destruct those member objects at the right time.
- So in addition to simply adding a dummy constructor, C++ also adds code to it to first call the default constructors of all member variables when the outer class is constructed (green)
- And when it adds a dummy destructor, C++ also adds code to the end of it to call the destructors of the member variables (in the reverse order of construction) when the outer object goes away (in red).
- This ensures that by the time your other member functions use your embedded object variables, they'll be properly initialized!
- And it ensures that when your outer object goes away, its embedded object variables will be destructed too!

Back to Auto-generated Constructors and Destructors

```
class HungryNerd
{
public:
  HungryNerd()  // implicitly added by compiler
    Call myBelly's  default constructor
    Call myBrain's default constructor
  {
    // empty
  }
  ~HungryNerd()  // implicitly added by compiler
  {
    // empty
  }
    Call myBrain's destructor
    Call myBelly's destructor
  void celebrate()
  {
    cout << "Happy " << myAge <<
         "'th birthday!";
  }
private:
    Belly myBelly;
    Brain myBrain;
    int   myAge;
```

- Just be careful!
- Remember, that the auto-added default constructor WON'T initialize your primitive member variables (e.g., ints, bools, floats, doubles, etc.)!
- In the class to the left, myAge would be such a primitive variable.
- So if your class has any primitives you should define your own constructor and initialize them!
- Otherwise they'll have essentially random values. Whatever was in the RAM before the variable was created will be the value of the variable.

Advanced Class Composition

Of course, things are never quite so simple…

Let's look at a slightly more complex class composition example.

Let's change the constructor of our Stomach class so it REQUIRES the user to specify the amount of gas each stomach starts with.

Advanced Cl ition

Ok, so now our new Stomach constructor REQUIRES us to pass in a value…

```
class Stomach
{
public:
  Stomach(int startGas)
  {
     myGas = startGas;
  }

  ~Stomach() { cout << "Fart!\n";  }
  void eat()  { myGas ++; }
private:
  int myGas;
};
```

```
int main()
{
     Stomach  a(5); // 5 farts
     Stomach b;  // ???

     a.eat();   // 6 farts
     …
}
```

- In the previous examples, when we used class composition, our member variables had default constructors (i.e., constructors that don't have/require any parameters).
- But what happens if one of our member variables has a constructor that REQUIRES parameters be passed to it.
- For example, take a look at this updated Stomach class that requires the amount of gas the stomach starts with to be passed in.
- As you see on the right, now we have to pass a value (like 5) in to construct a new Stomach variable.
- Stomach no longer has a default constructor to call!
- Now, what happens if we want to use our new Stomach class in our HungryNerd class?

Advanced Class Composition

```
class Stomach
{
public:
   Stomach(int startGas)
   {
       myGas = startGas;
   }
   ~Stomach() { cout << "Fart!\n";  }
   void eat()  { myGas ++; }
private:
  int myGas;
};
```

```
class HungryNerd
{
public:
   HungryNerd()
   {
       myBelly.eat();
       myBrain.think();
   }


   ...

private:
   Stomach myBelly;
   Brain myBrain;
};
```

Will our HungryNerd class still compile?

NO!

Advanced Class Composition

```
class Stomach
{
public:
   Stomach(int startGas)
   {
      myGas = startGas;
   }
   ~Stomach() { cout << "Fart!\n";  }
   void eat()  { myGas ++; }
private:
  int myGas;
};
```

```
class HungryNerd
{
public:
   HungryNerd()
      Call myBelly's default constructor
      Call myBrain's default constructor
   {
      myBelly.eat();
      myBrain.think();
   }
   ...

private:
   Stomach myBelly;
   Brain myBrain;
};
```

- The auto-generated code that C++ adds to construct the member variables (green box on right) before the outer constructor runs always calls default constructors, which take NO parameters!
- But our new Stomach class doesn't have a default (parameter-less) constructor anymore! If you want to construct a Stomach variable you **have** to pass in a value to its constructor.
- So this results in a C++ compilation error, since there's no default constructor in Stomach for C++ to call in your HungryNerd constructor.
- In these cases (where a member variable requires a parameter to be constructed), we have to explicitly add some code to our constructor to call the member variable's constructor and pass in valid parameters.

Advanced Class Composition

```
class Stomach
{
public:
   Stomach(int startGas)
   {
      myGas = startGas;
   }
   ~Stomach() { cout << "Fart!\n";  }
   void eat()  { myGas ++; }
private:
  int myGas;
};
```

```
class HungryNerd
{
public:
   HungryNerd()
     : myBelly(10)
   {
      myBelly.eat();
      myBrain.think();
   }

private:
   Stomach myBelly;
   Brain myBrain;
};
```

We add an "initializer list" to explicitly initialize member variables whose c'tors require one or more parameters.

Call myBrain's default constructor

- Alright, let's see the "exciting" C++ syntax!
- We're no longer going to let C++ call Stomach's default constructor for us, since Stomach doesn't have a default c'tor!
- Instead, we'll add our own C++ code to explicitly call myBelly's constructor with a parameter.
- Notice the new code in green/red that the programmer has to add between the outer constructor's prototype and the { that starts its code block:

  `: myBelly(10)`
- This new code is called an "initializer list". We use an initializer list to explicitly construct the myBelly variable by passing in a value of 10. Now C++ won't try to call the default constructor.
- The initializer list explicitly calls the member variable's constructor and passes in the required values.
- As before, we can still rely upon C++ to implicitly call the constructor for myBrain (green box), since it still has a default constructor.

Advanced Class Composition

```
class Stomach
{
public:
   Stomach(int startGas)
   {
      myGas = startGas;
   }
   ~Stomach() { cout << "Fart!\n";  }
   void eat()  { myGas ++; }
private:
  int myGas;
};
```

and its body.

```
class HungryNerd
{
public:
   HungryNerd()
     : myBelly(10)
   {
      myBelly.eat();
      myBrain.think();
   }

private:
   Stomach myBelly;
   Brain myBrain;
};
```

Call myBrain's default constructor

- Any time you have a member variable (e.g., myBelly) that requires one or more parameters for construction…
- You must add an initializer list to *all* of your outer class's constructor(s).
- The initializer list sits between the constructor's prototype (HungryNerd() in this case), and its open bracer {.
- The initializer list starts with a colon, followed by one or more member variables and their parameters in parentheses.

Advanced Class Composition

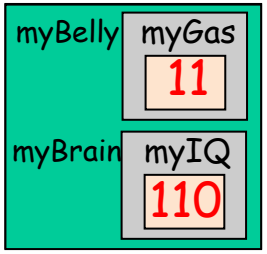
```
class Stomach
{
public:
   Stomach(int startGas)
   {
      myGas = startGas;
   }
   ~Stomach() { cout << "Fart!\n";  }
   void eat()  { myGas ++; }
private:
  int myGas;
};
```

```
class Brain
{
public:
    Brain()       { myIQ = 100; }
    void think() { myIQ += 10; }

};
```

```
class HungryNerd
{
public:
   HungryNerd()
     : myBelly(10)
       Call myBrain's default constructor
   {
      myBelly.eat();
      myBrain.think();
   }

private:
   Stomach myBelly;
   Brain myBrain;
};
```

- So when we construct our HungryNerd (carey), the first thing that happens is HungryNerd's initializer list calls the constructor for its myBelly Stomach variable and passes in 10.
- Then the implicitly added code calls myBrain's default constructor (as it did before)
- Finally, HungryNerd's constructor runs last, allowing carey to .eat() and .think().

carey




```
int main()
{
    HungryNerd   carey;

    ...
}
```

Advanced Class Composition

```
class Stomach
{
public:
   Stomach(int startGas)
   {
      myGas = startGas;
   }
   ...
};
```

```
class Brain
{
public:
    Brain(int startIQ)
    {
       myIQ = startIQ;
    }
    void think() { myIQ += 10; }
};
```

```
class HungryNerd
{
public:
   HungryNerd()
     : myBelly(10) , myBrain(150)

   {
      myBelly.eat();
      myBrain.think();
   }

private:
   Stomach myBelly;
   Brain myBrain;
};
```

```
int main()
{
   HungryNerd   carey;

   ...
}
```

- We can also use the initializer list to initialize more than just one member variable!
- For instance, let's assume we changed our Brain class so it requires the initial IQ value to be passed in (see just above)
- Now we would be required to update our initializer list to explicitly initialize both myBelly (in this example, with a value of 10), and myBrain (in this example, with an initial IQ of 150)
- Notice that we separate each item in the initializer list with a comma.

Initializer Lists

- If you like, you may also use the initializer list to initialize your primitive member variables too.
- For instance, in the example below, our HungryNerd now has a myAge integer member variable
- We now use the initializer list to set its value to 19. This ensures the myAge variable is initialized before the actual { body } of the constructor runs.
- You can initialize these primitive member variables either using the initializer list or just inside the body of the constructor as we've done in the past: myAge = 19;
- It's up to you (or the company you work for) to pick an approach and standardize on it.

```
class HungryNerd
{
public:

 HungryNerd()
  : myBelly(10), myBrain(150), myAge(19)
 {
   myBelly.eat();

 }

private:
  Belly myBelly;
  Brain myBrain;
  int   myAge;
};
```

Advanced Class Composition

```
class Stomach
{
public:
   Stomach(int startGas)
   {
      myGas = startGas;
   }
   ~Stomach() { cout << "Fart!\n";  }
   void eat()  { myGas ++; }
private:
  int myGas;
};
```

- Finally, there's no reason why the parameters to your initializer list have to be constants…
- You can pass any value or expression you like to your variables in the initializer list.
- In this example, we pass in the starting amount of gas to our HungryNerd constructor, and then our initializer list passes this value * 4 along to the Stomach constructor when constructing myBelly.
- So our myBelly variable would start out with 300 units of gas.
- Oh man, that must smell.

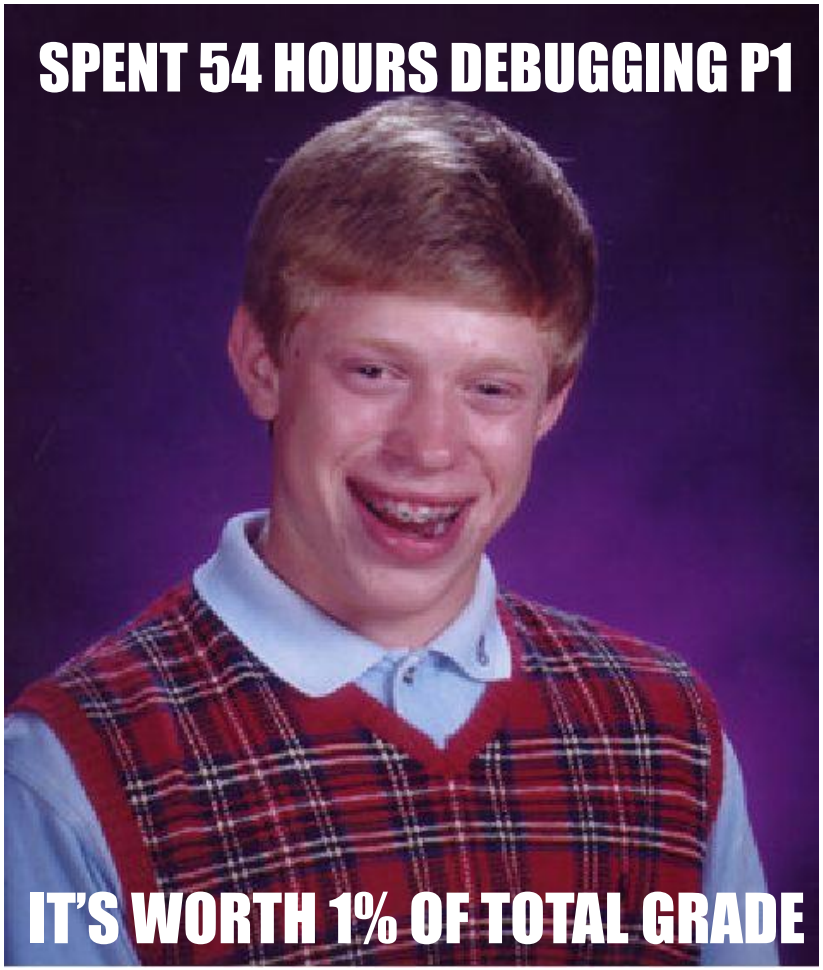
```
class HungryNerd
{
public:
   HungryNerd( int startingGas )
     : myBelly( startingGas*4 )
       Call myBrain's default constructor
   {
      myBelly.eat();
      myBrain.think();
   }

private:
   Stomach myBelly;
   Brain myBrain;
};
```

```
int main()
{
   HungryNerd   carey(75);

   ...
}
```

A Few Final Topics

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

Debugging

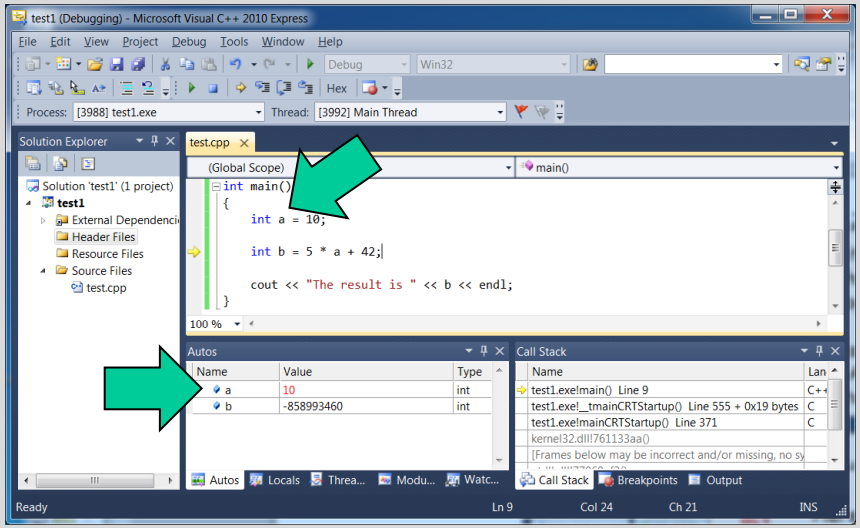


Learning to Use the C++ Debugger

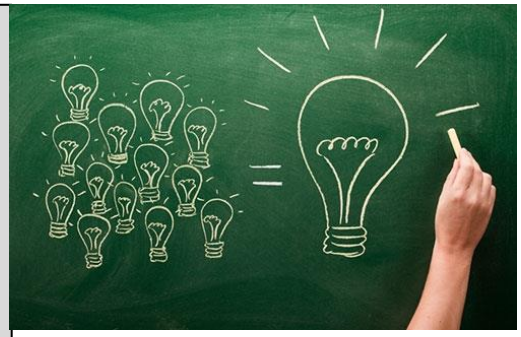
What's the big picture?

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!

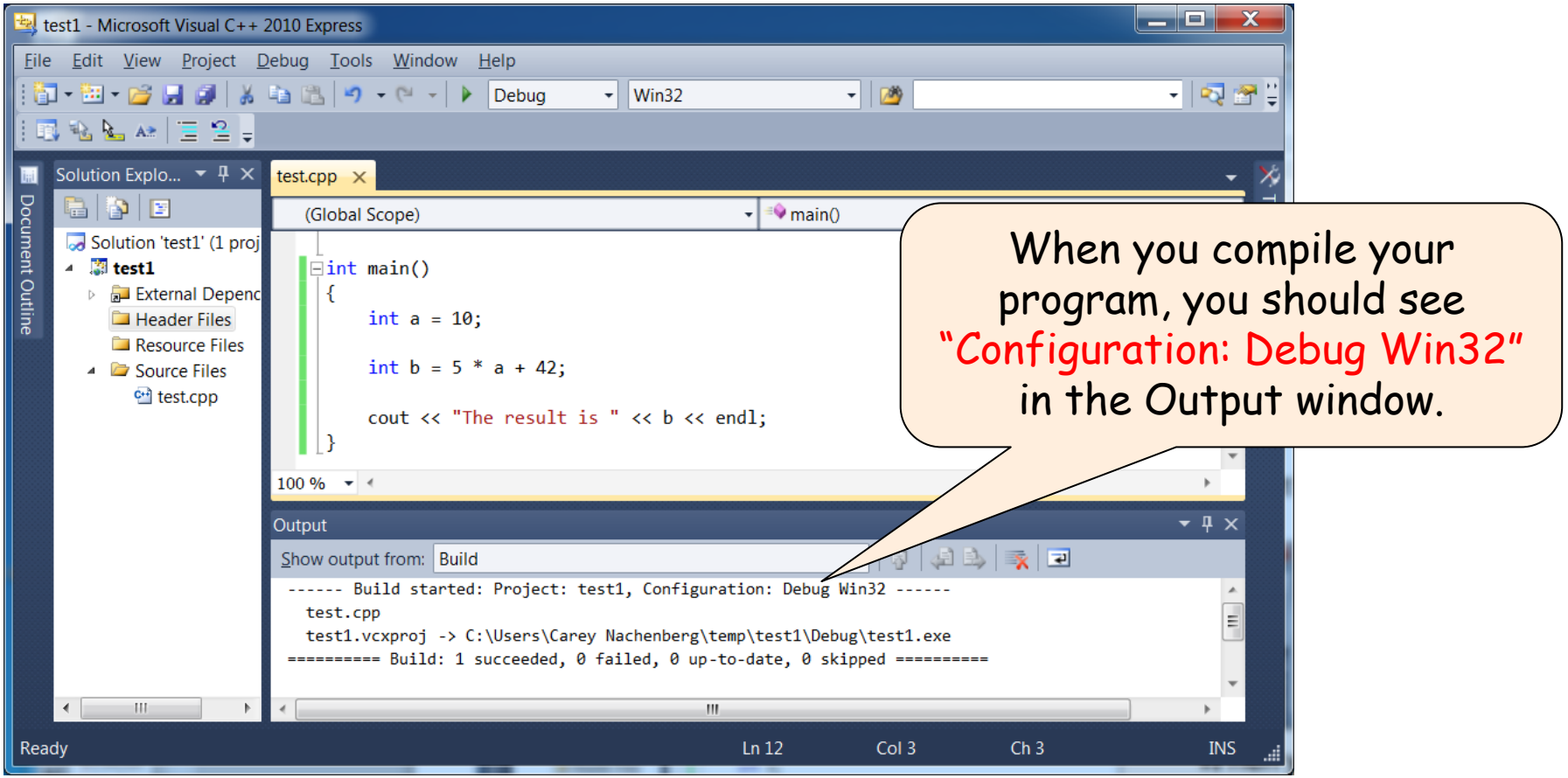


Uses:

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

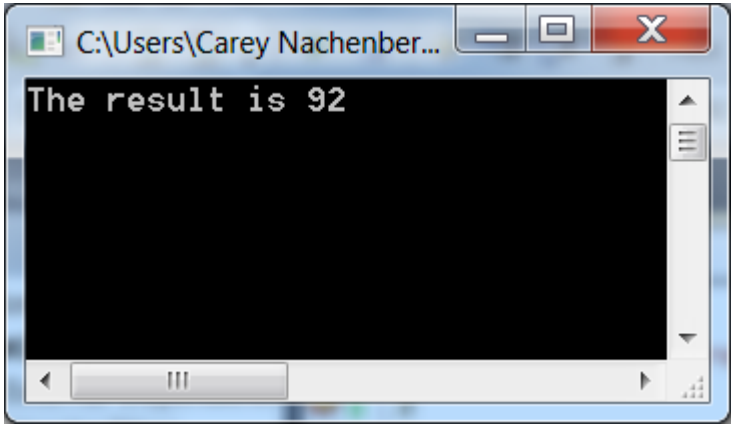
Debugging

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

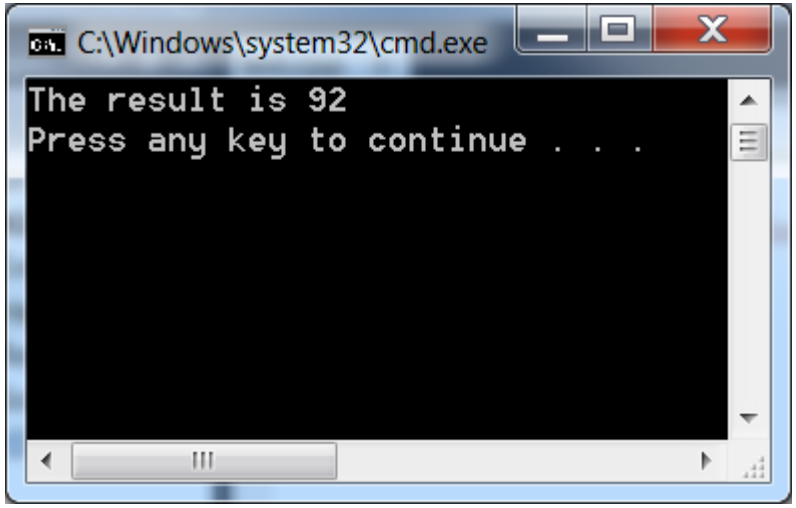


Debugging

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



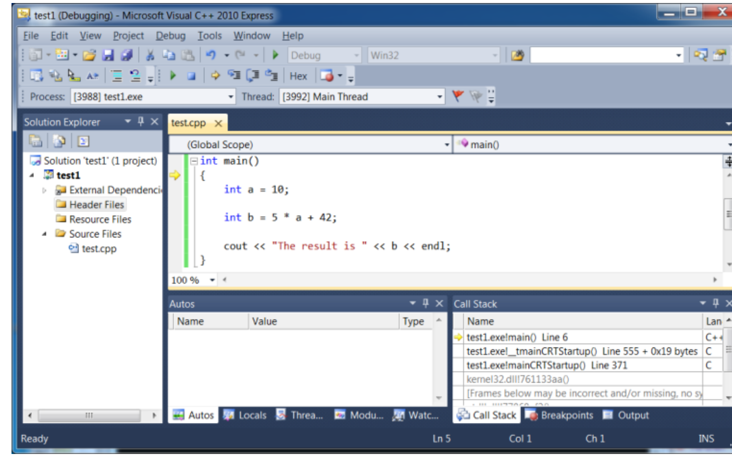
If you run your program by hitting F5, Visual Studio will immediately close the debug window after your program finishes.

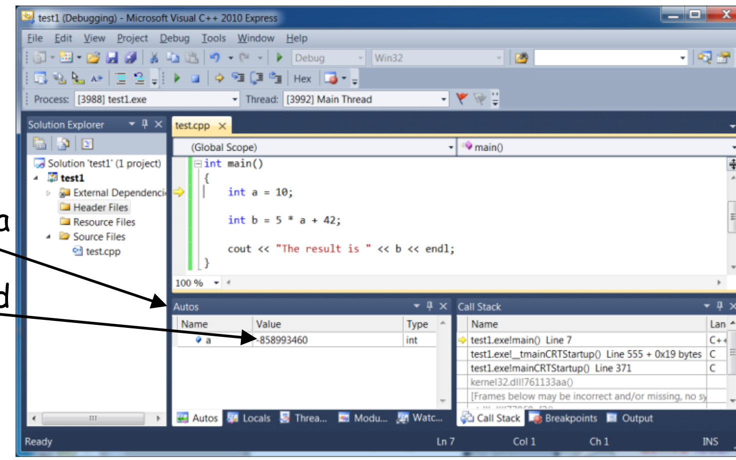


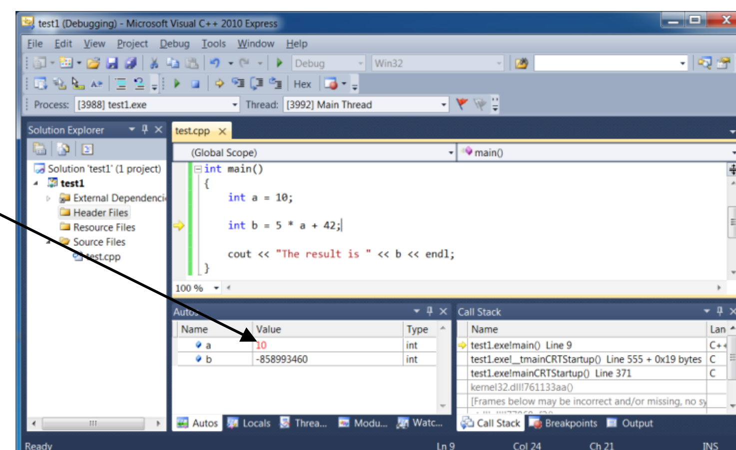
If you run by hitting Ctrl-F5, Visual Studio will print "Press any key to continue..." and leave the debug window on the screen when your program completes.

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!

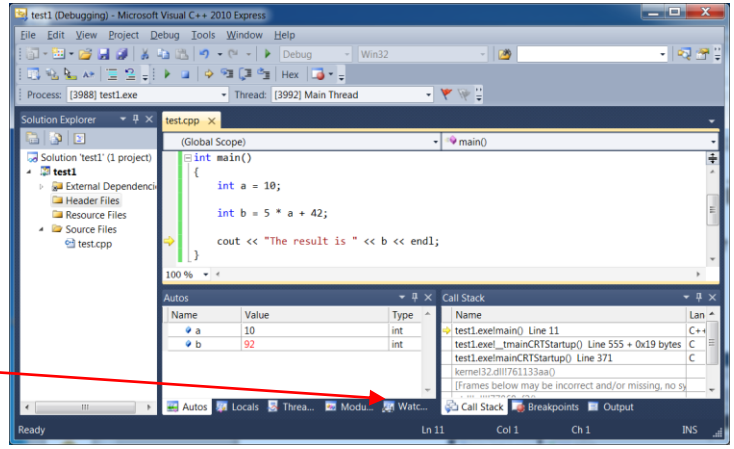


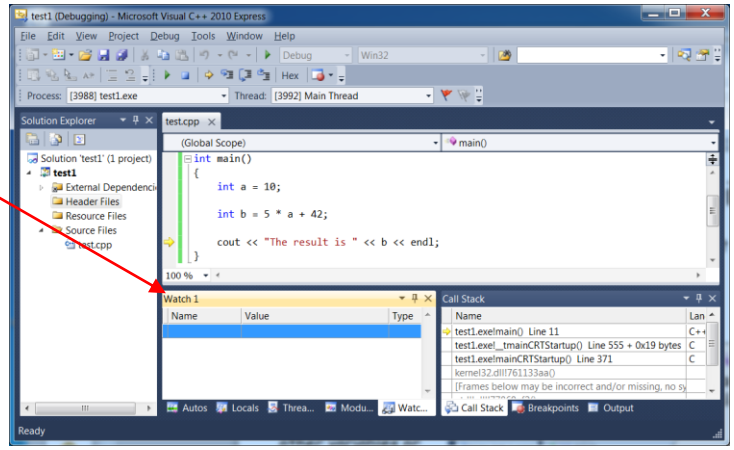


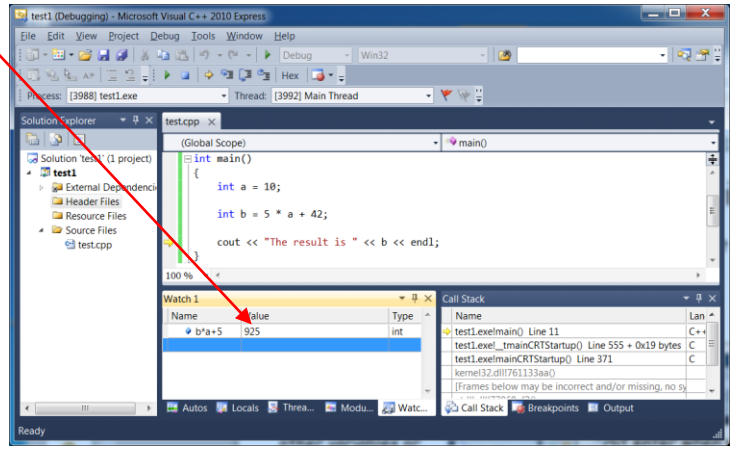


Debugging

- 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!

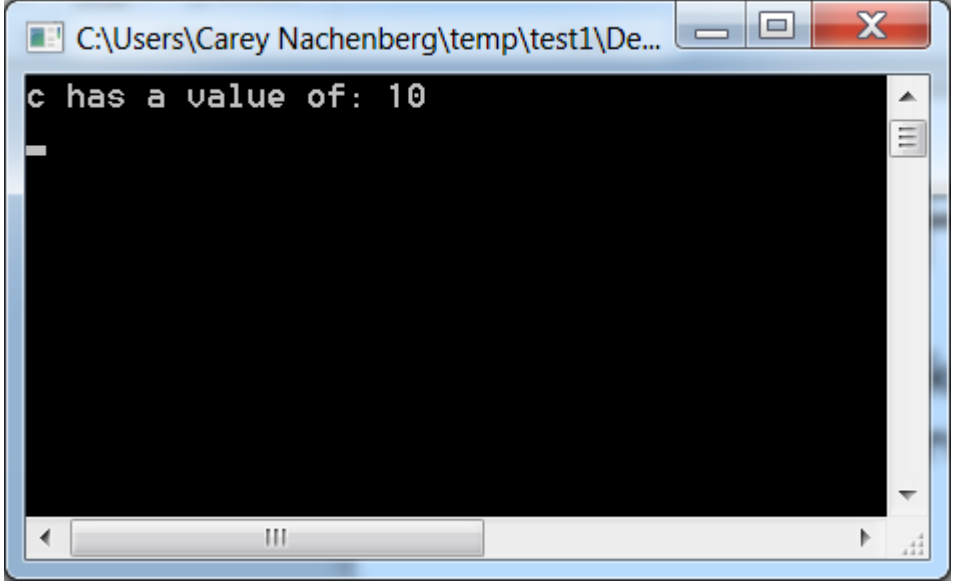


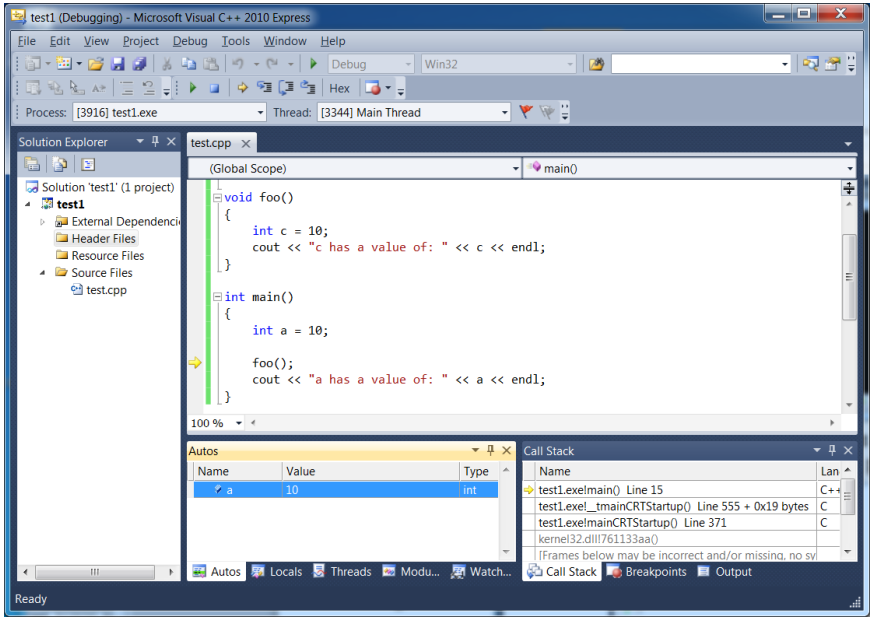


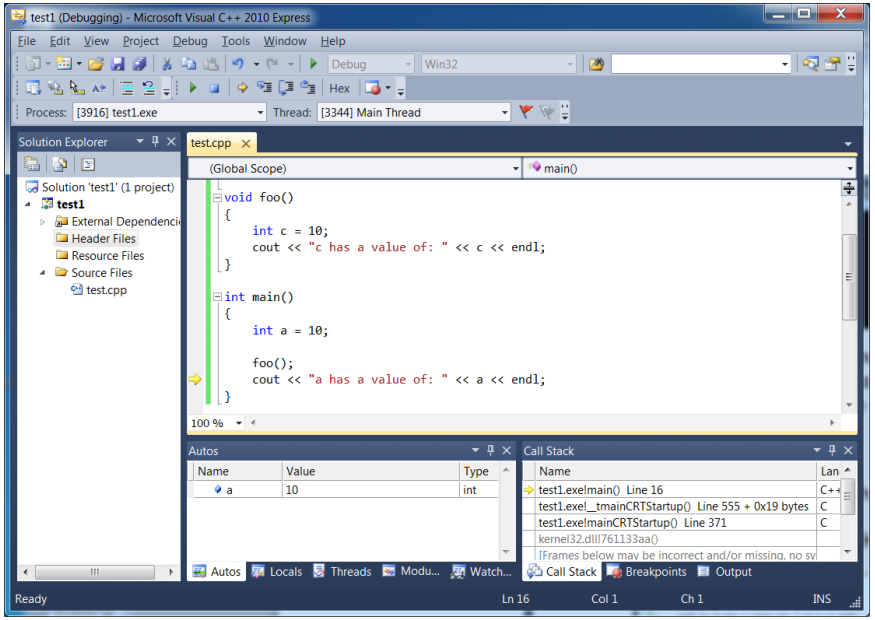


Debugging

- 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.

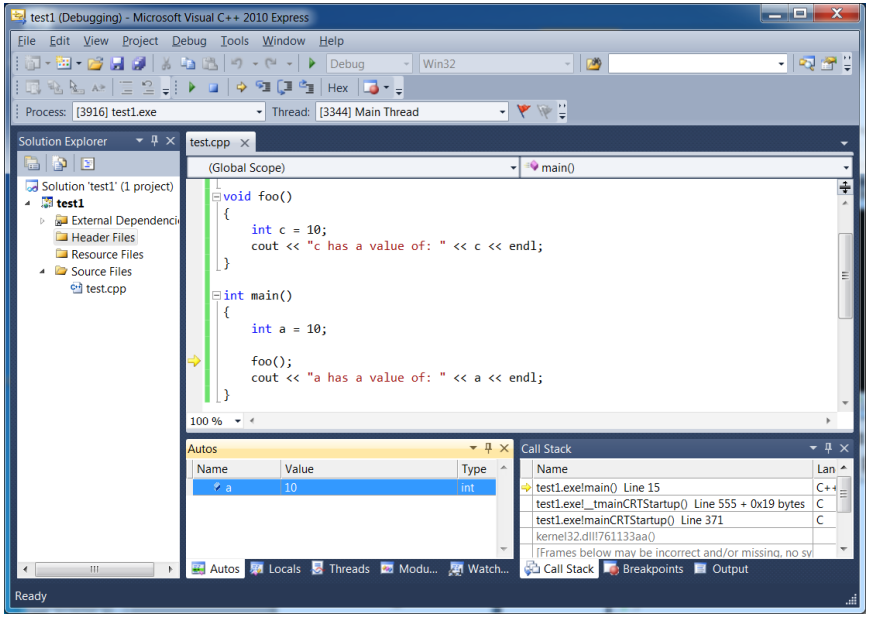


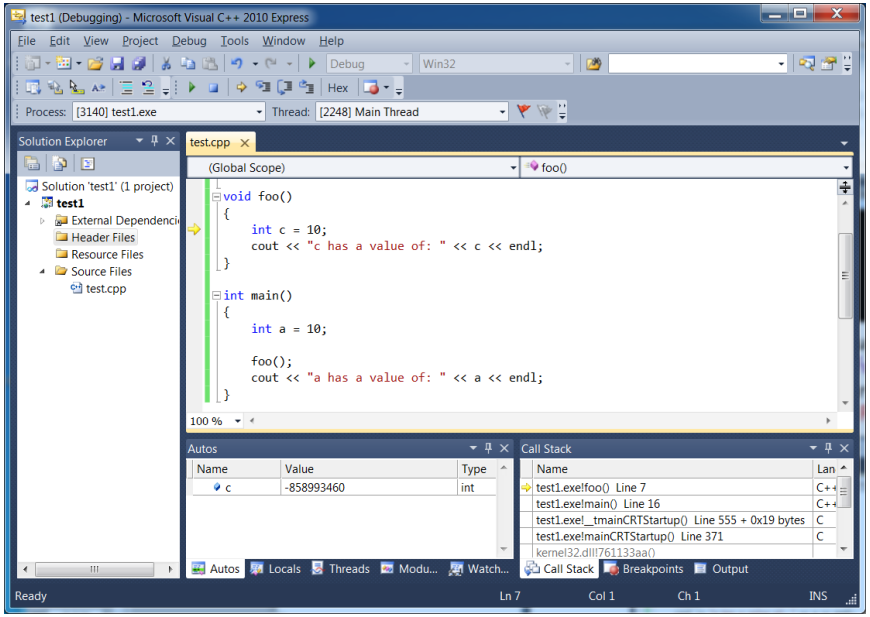




Debugging

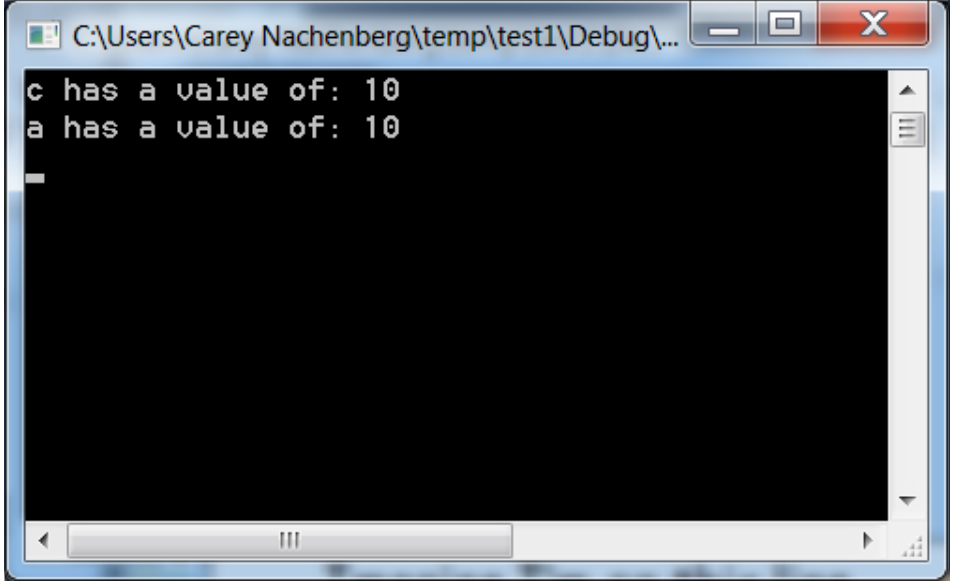
- Let's discuss F11 next. F11 means "TRACE INTO"
- If you hit F11 while on a line that contains a function call, Visual Studio will let you step *into* that function and trace through that function's statements a line at a time.
- So let's say you're debugging your program and are about to call the foo() 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.

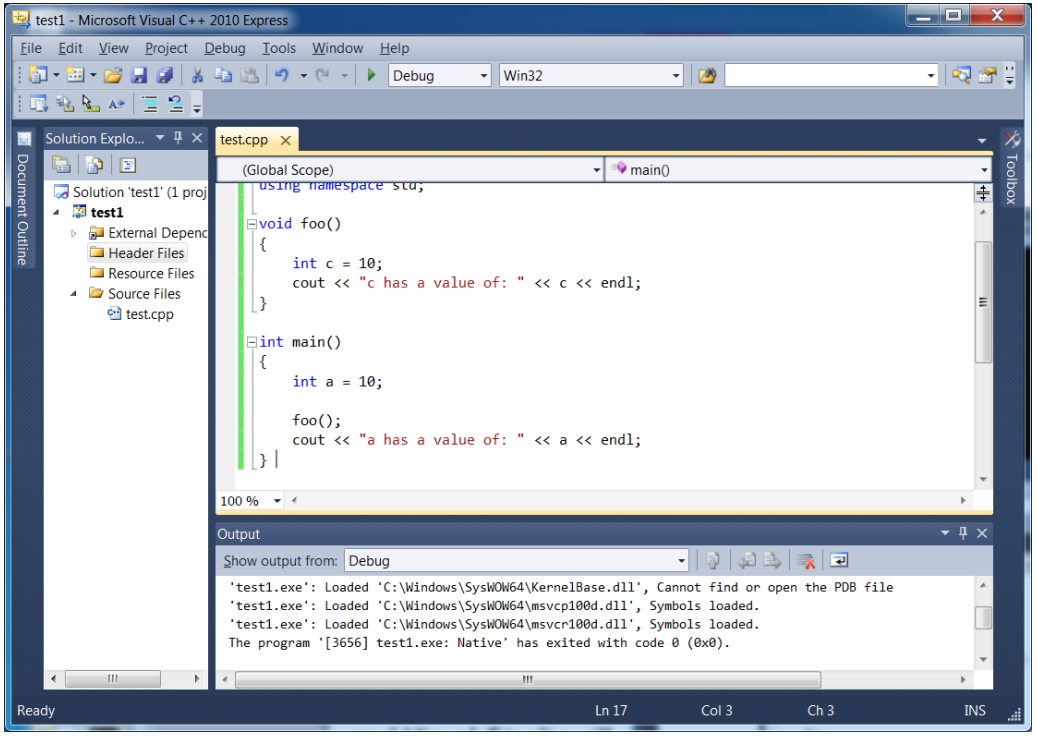




Debugging

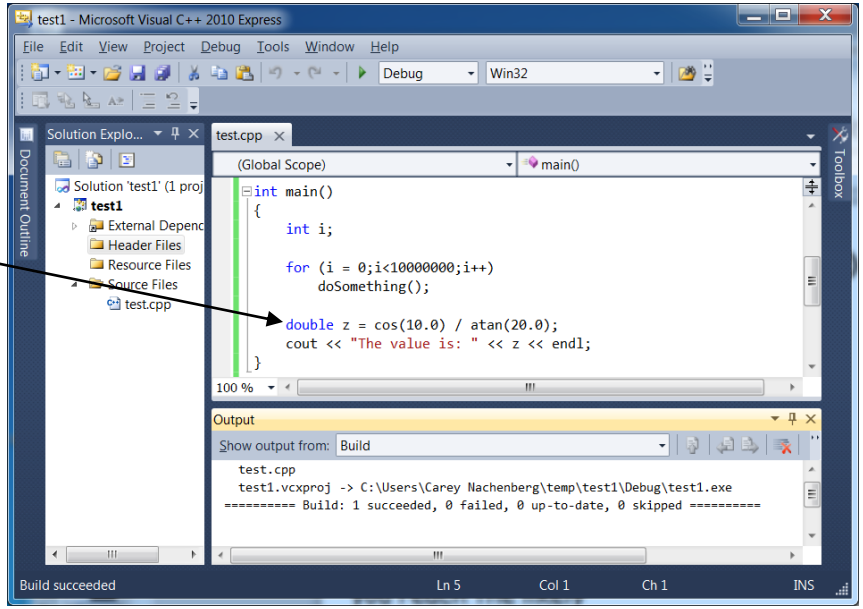
- If at any time while you're tracing line-by-line you want to stop tracing line-by-line…
- And simply let the rest of your program run normally…
- Just hit F5 and Visual Studio will run your program until completion!
- Be careful… your output window will immediately disappear after the program finishes 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.

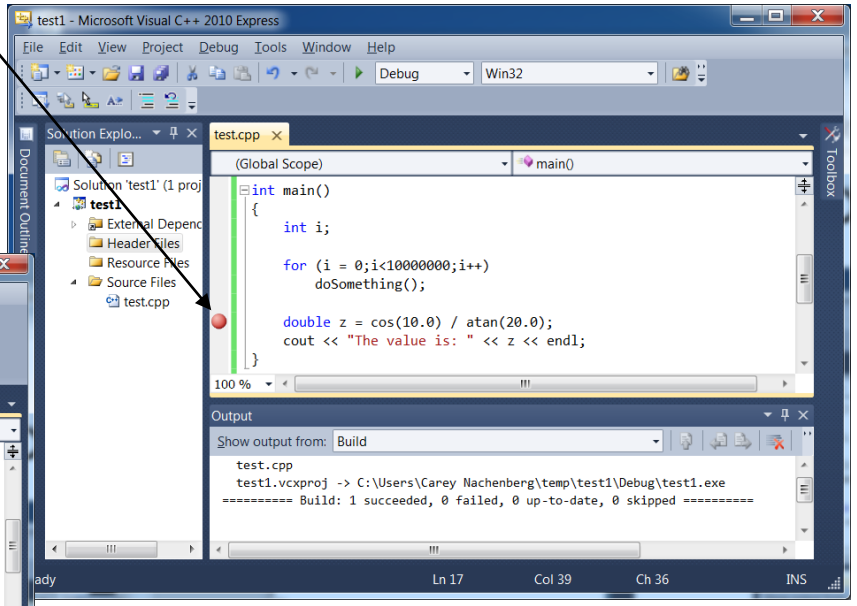


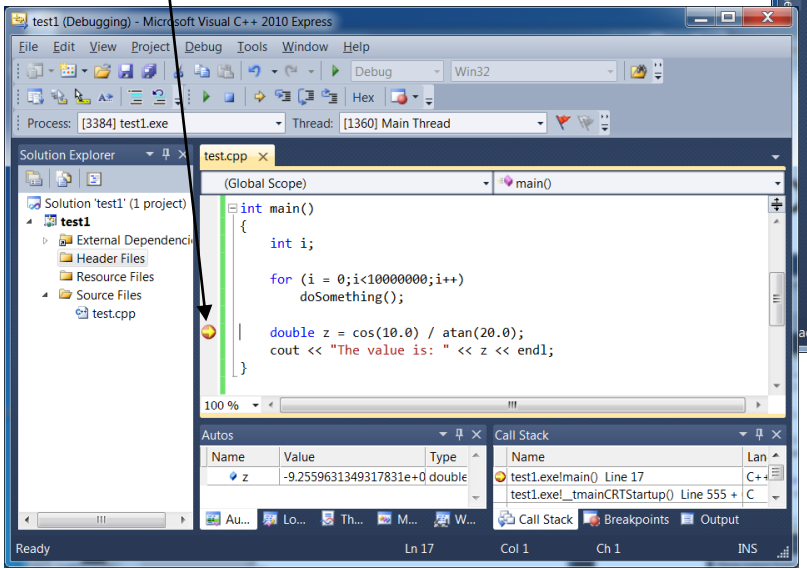


Debugging

- 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!

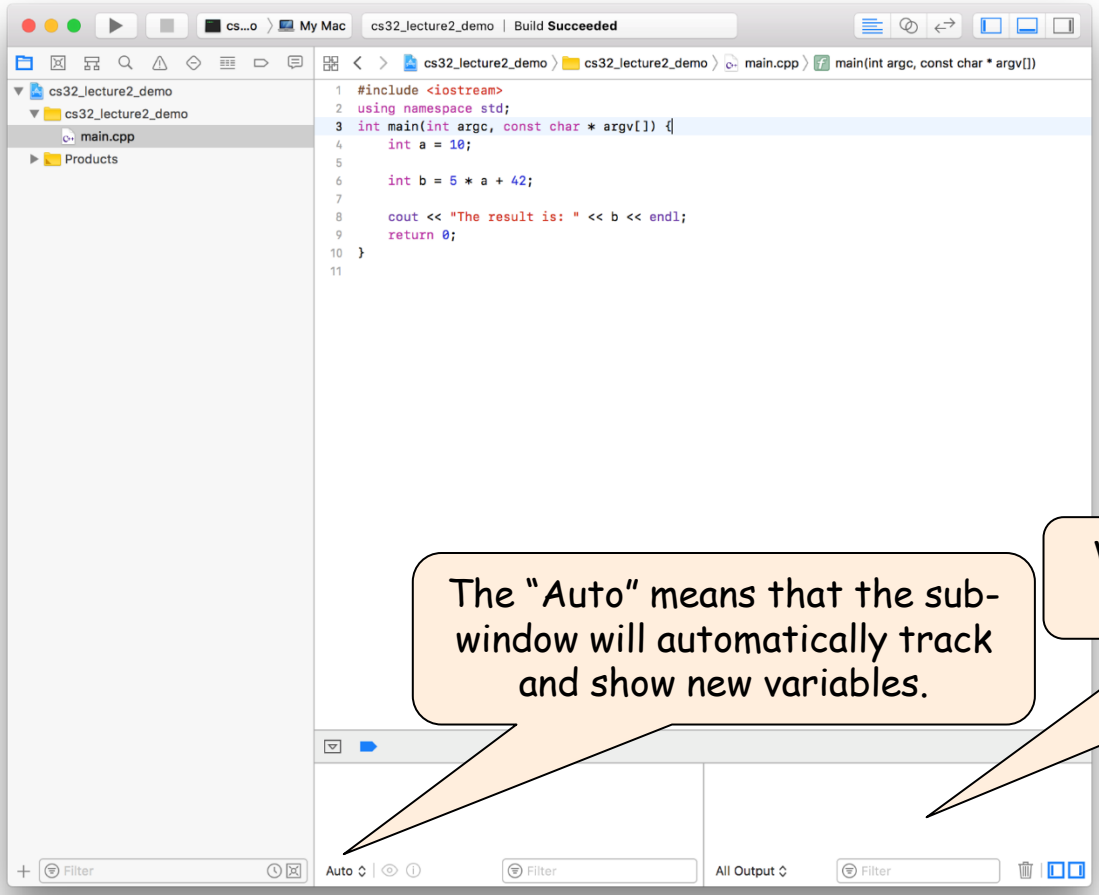
Debugging with Xcode

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**.



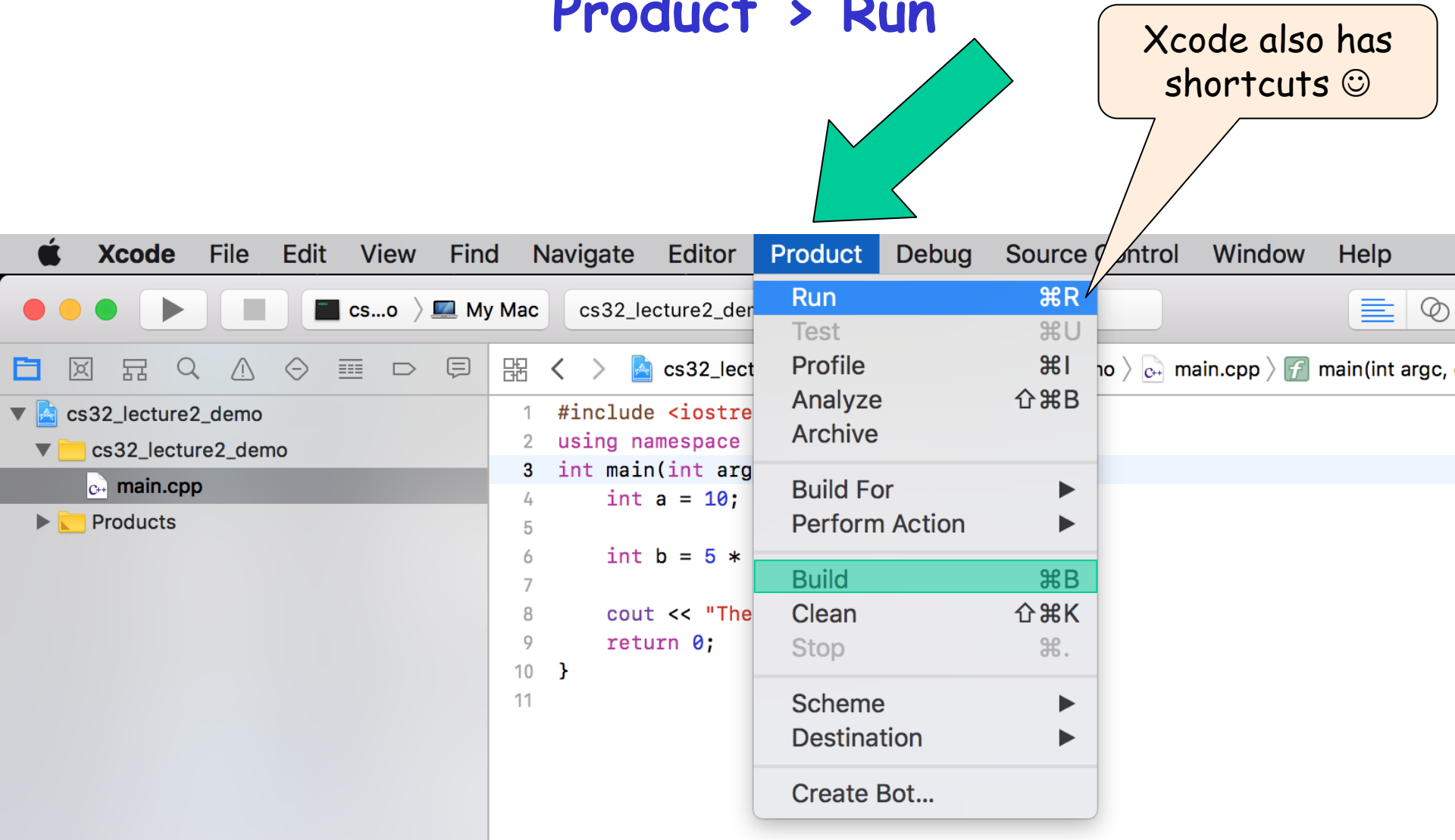
The "Auto" means that the sub-window will automatically track and show new variables.

When you compile your program, you'll see an empty debug output window…

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 }
```

```
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
```

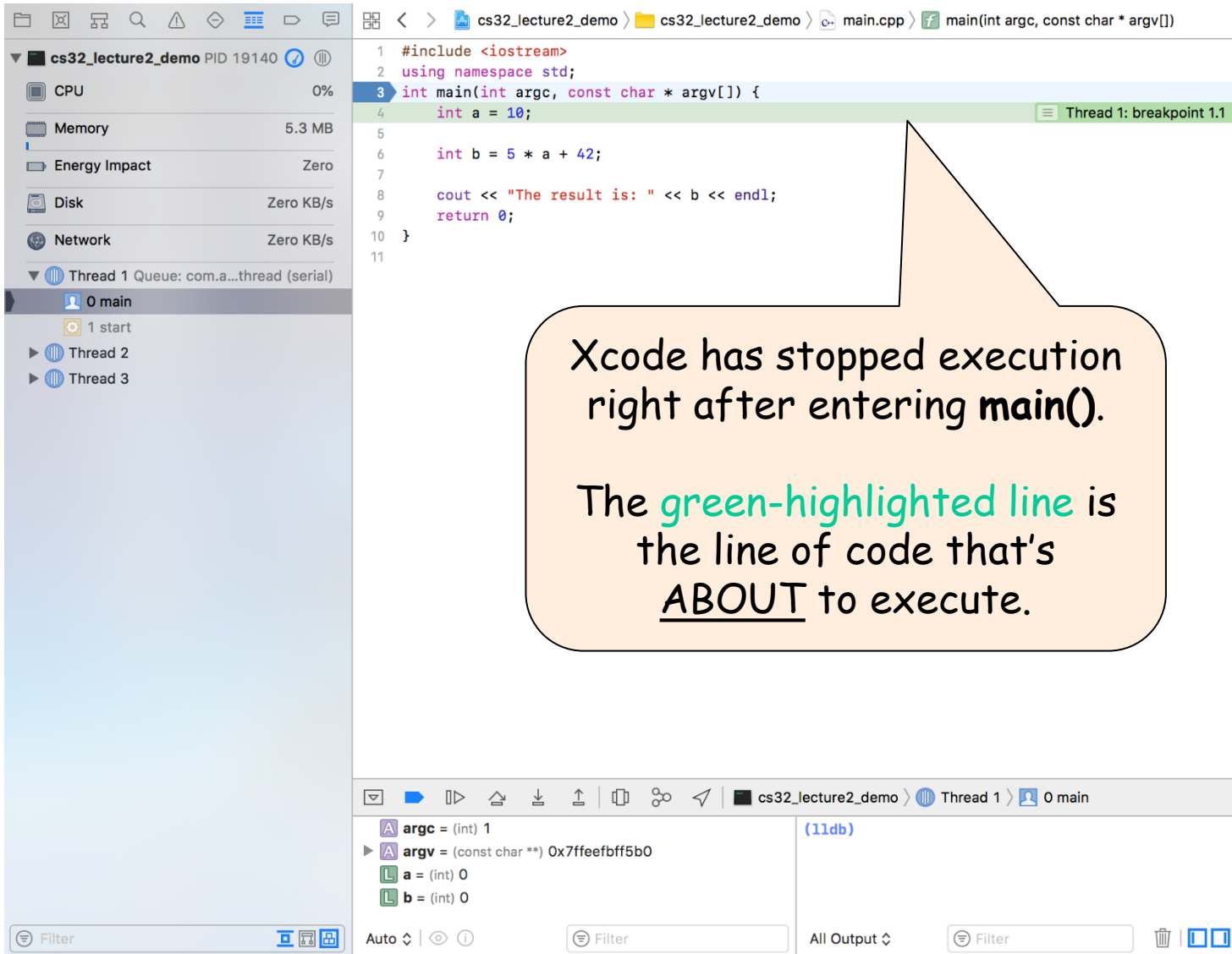
Debugging with Xcode

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.

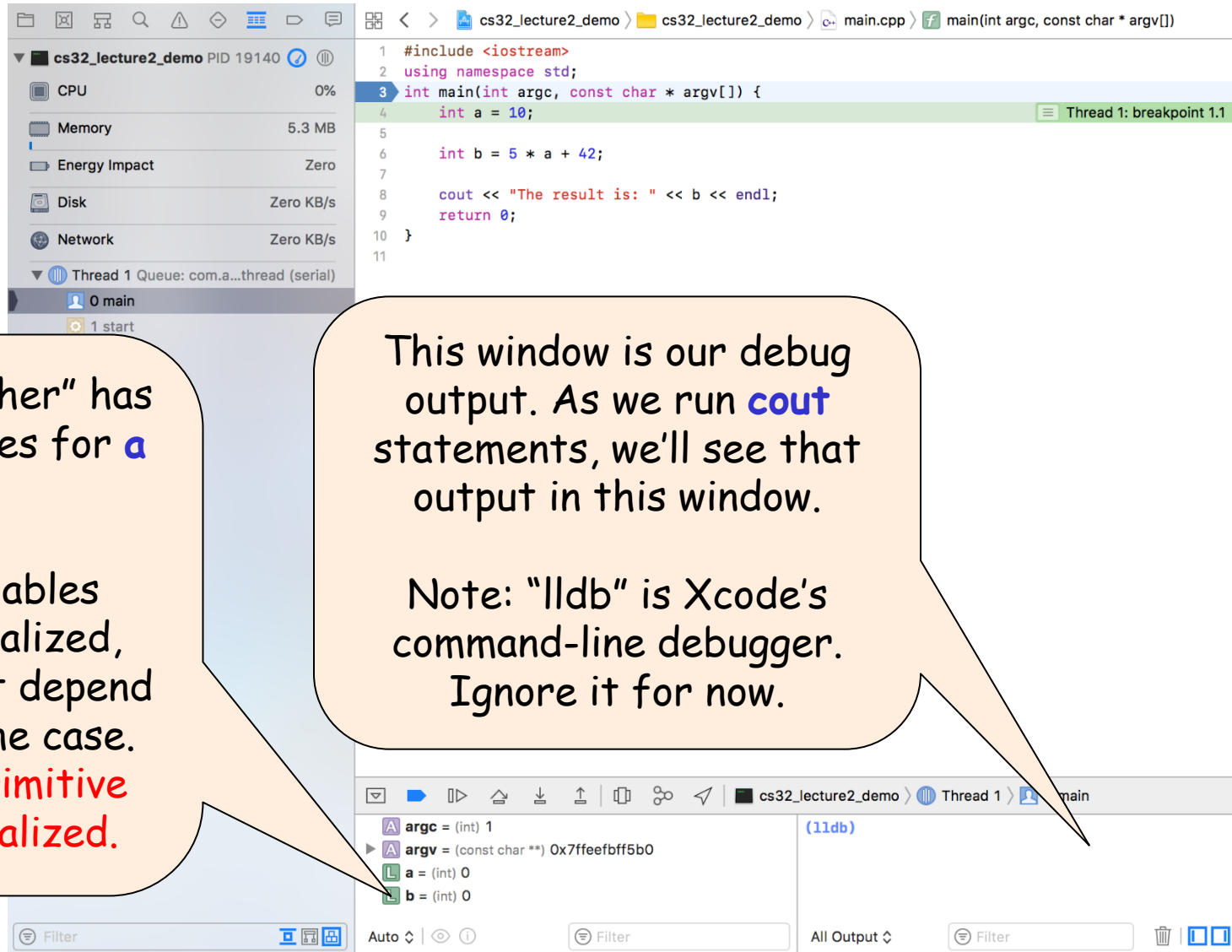
The left pane is showing resource usage.

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**.

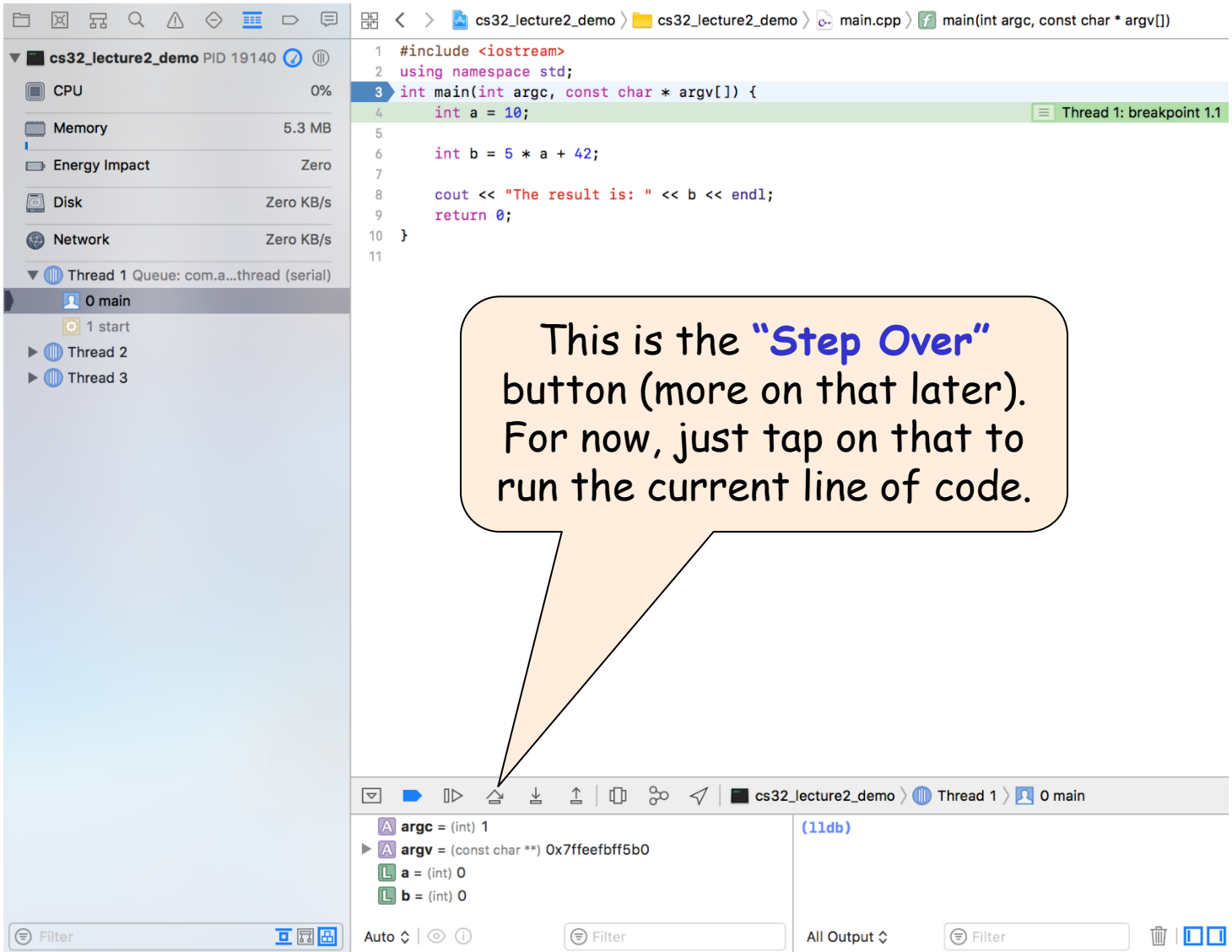
The a and b variables appear to be initialized, but in C++ you can't depend upon that to be the case. Always assume primitive values are uninitialized.

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.

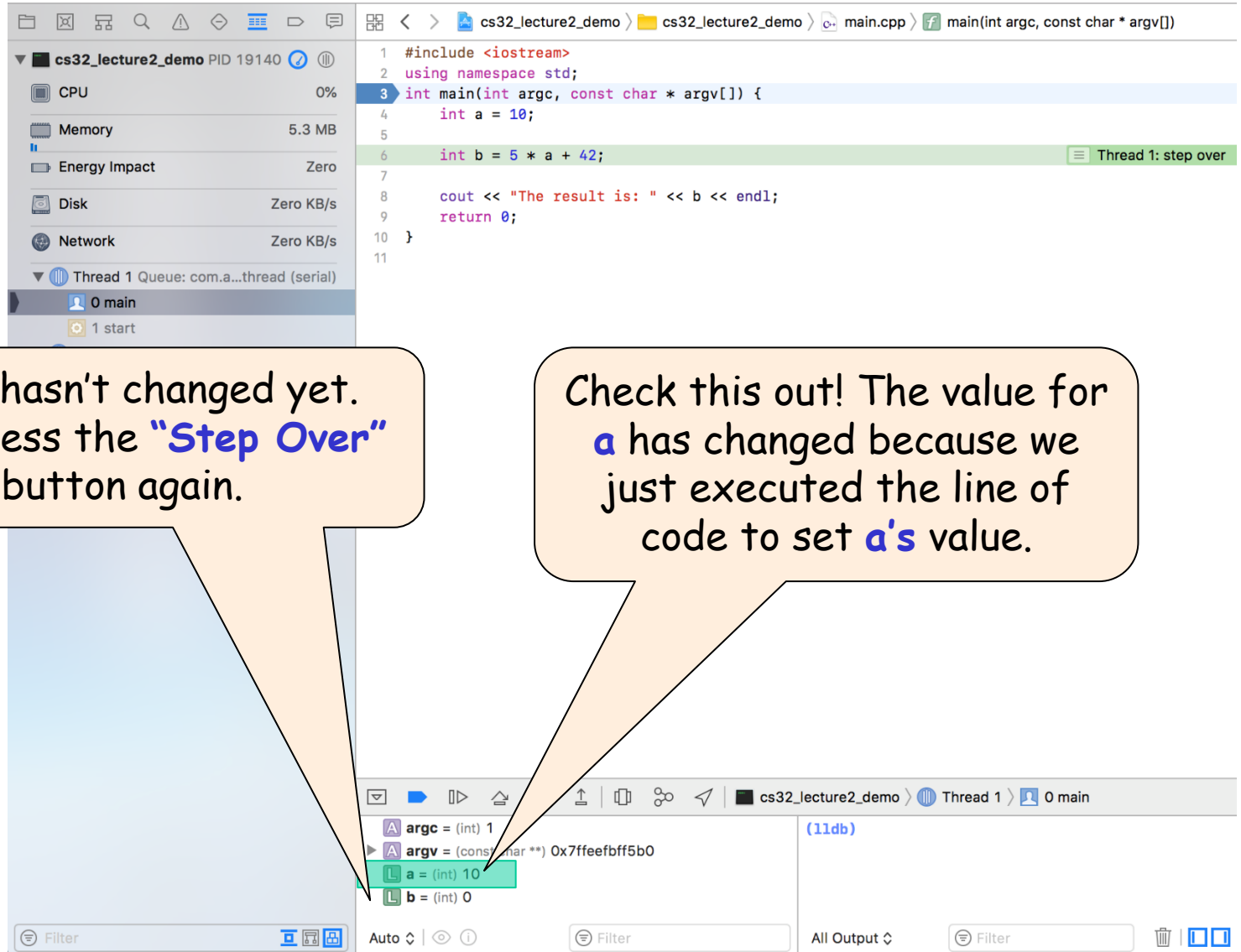
Debugging with Xcode

Let's run the current line of code.



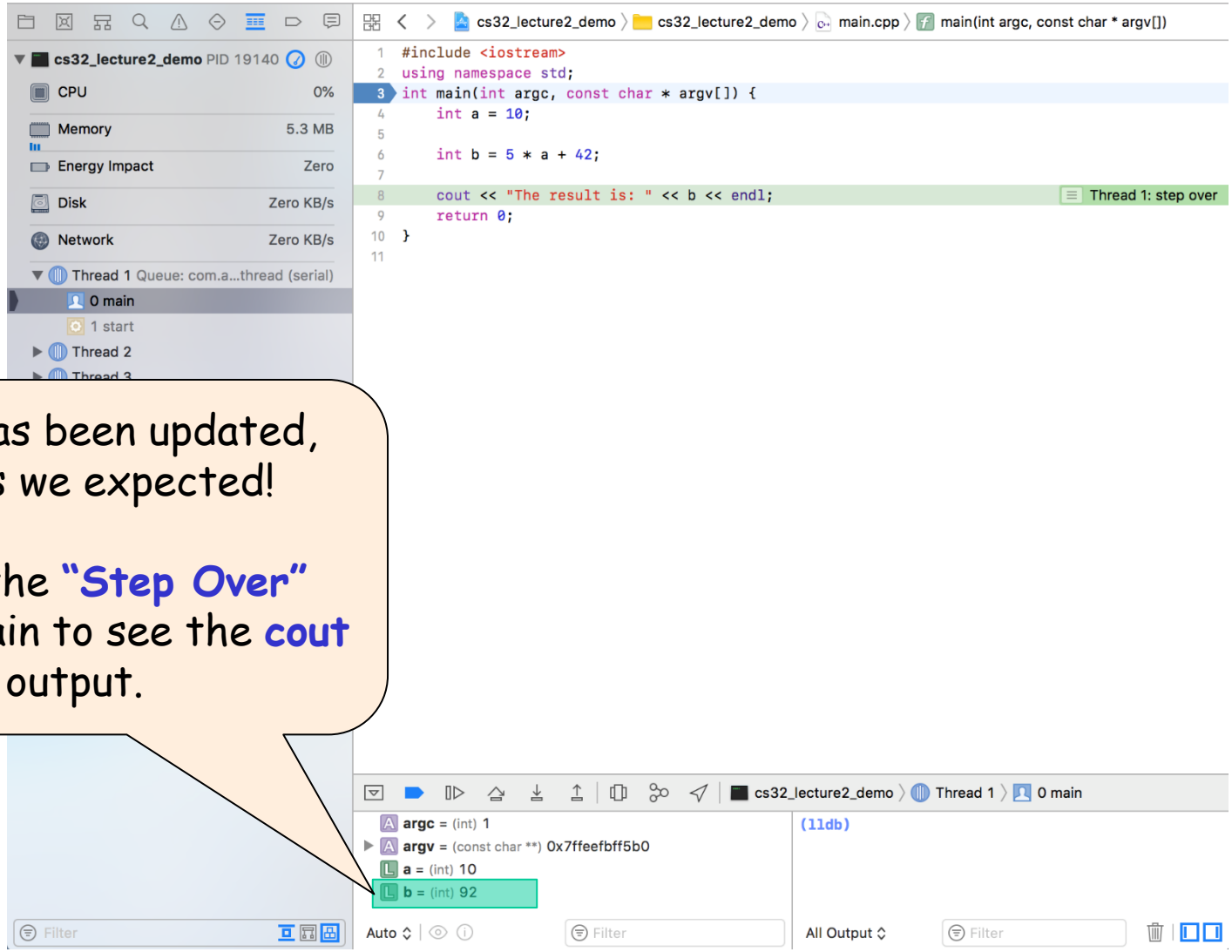
Debugging with Xcode

Let's run the current line of code.



Debugging with Xcode

Let's run the current line of code.

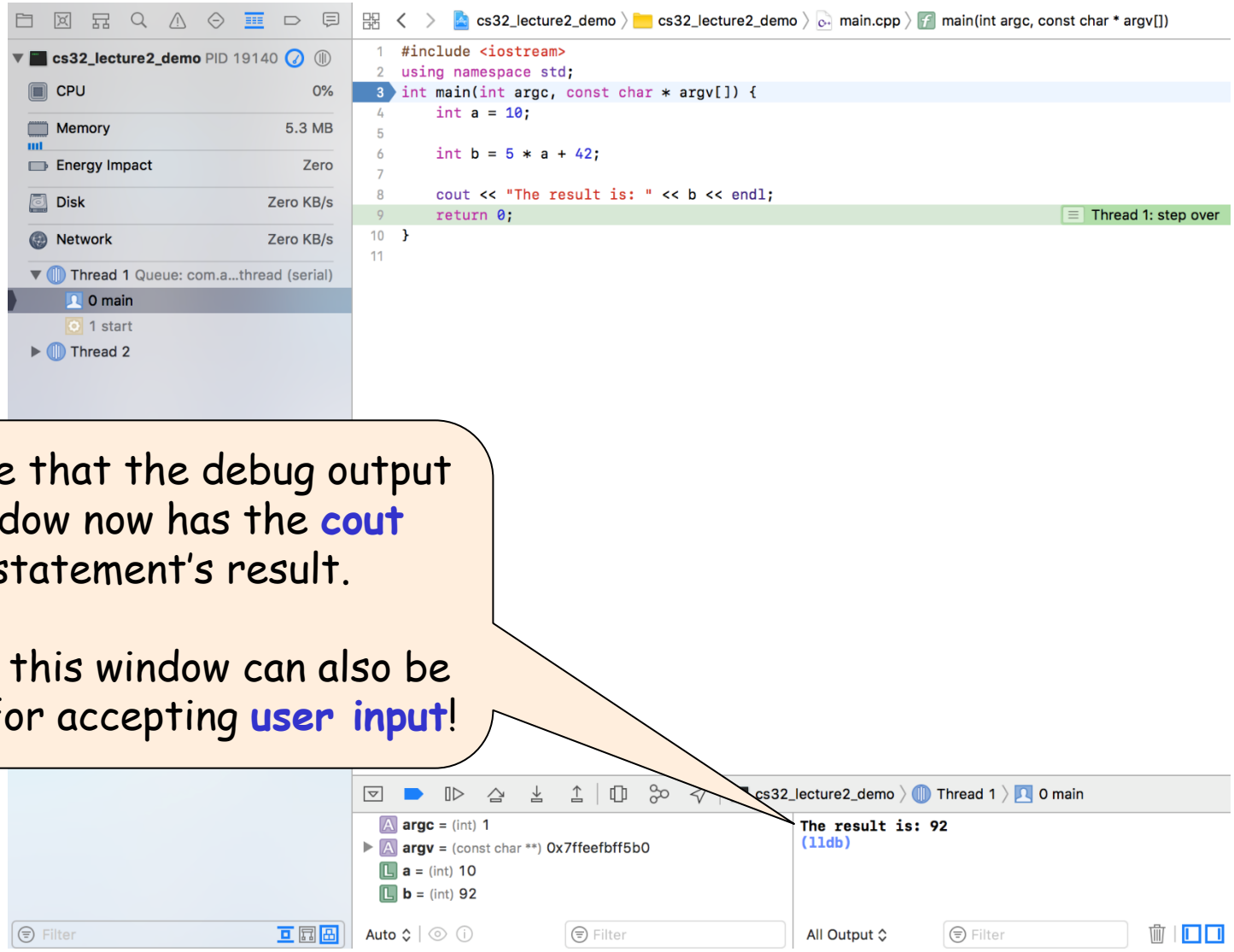


Now **b** has been updated, just as we expected!

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

Debugging with Xcode

Let's run the current line of code.



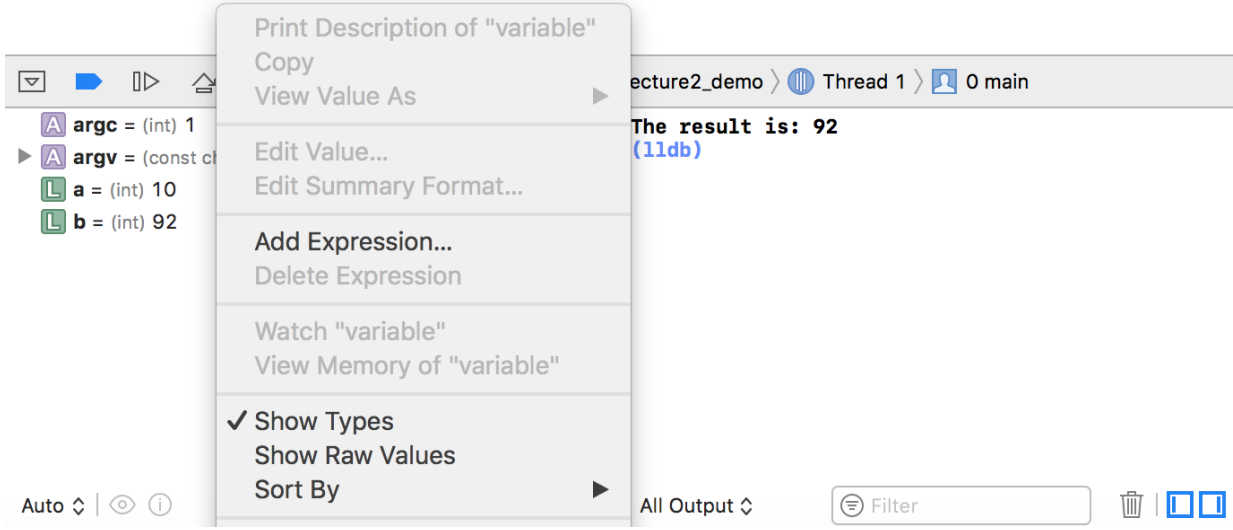
Notice that the debug output window now has the **cout** statement's result.

BTW, this window can also be used for accepting **user input**!

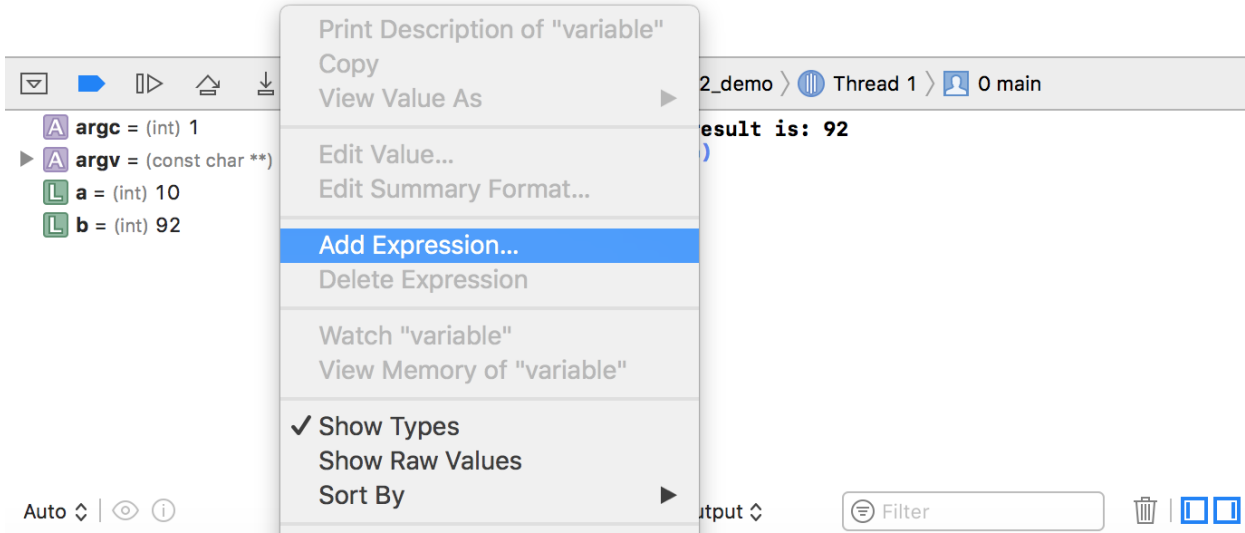
Debugging with Xcode

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

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

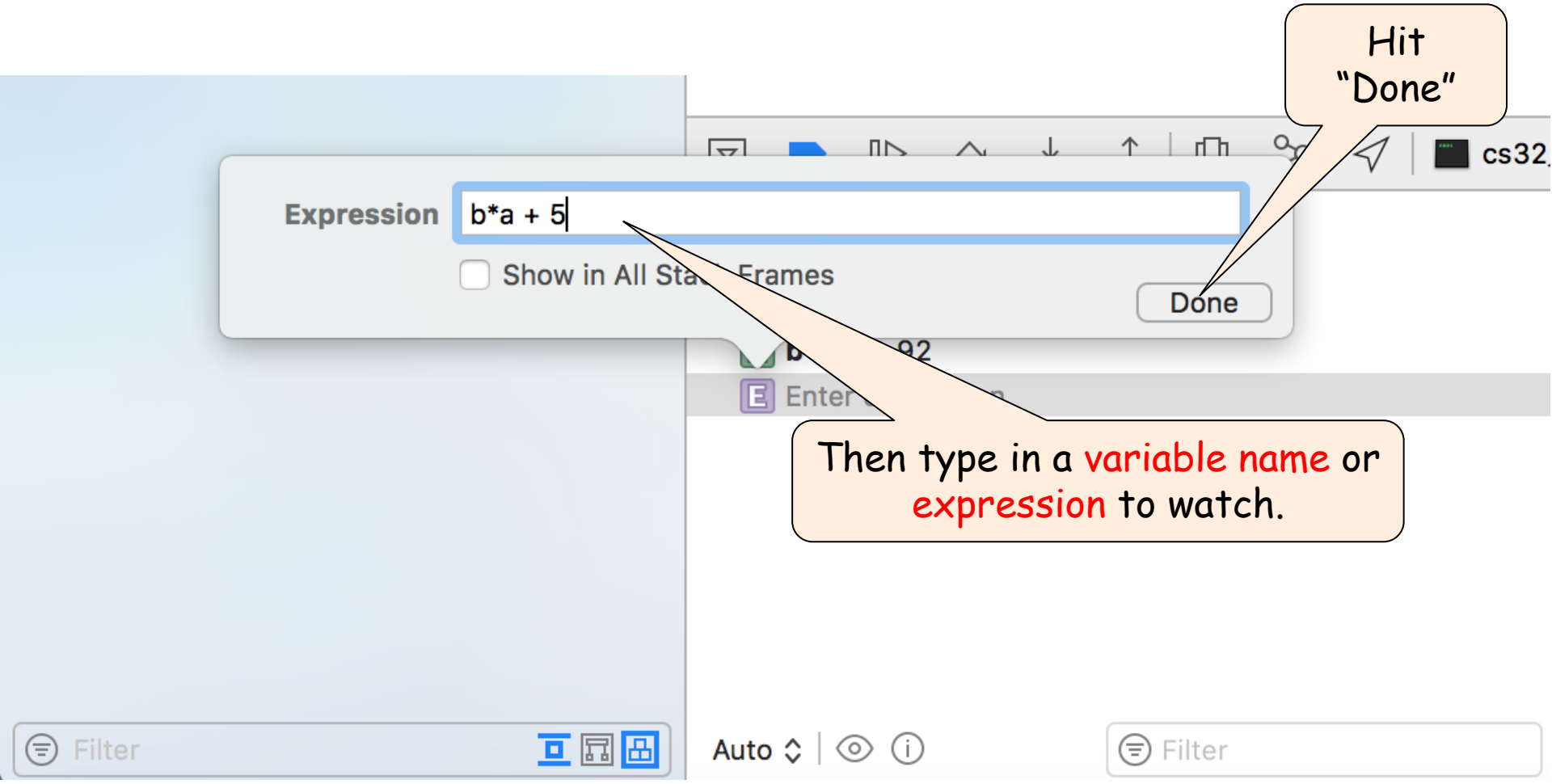


Now click on ***Add Expression...***



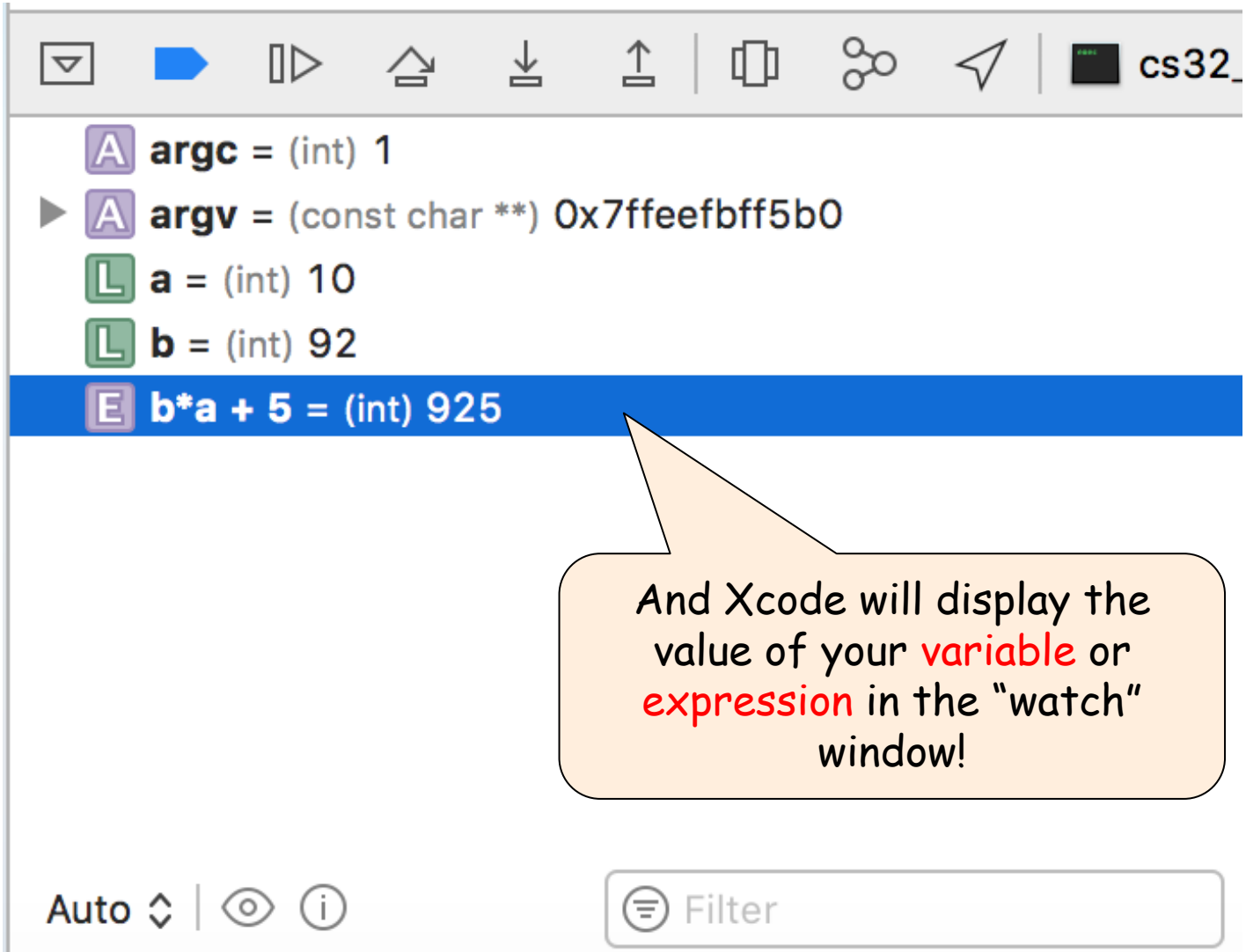
Debugging with Xcode

Add an expression to watch



Debugging with Xcode

Add an expression to watch



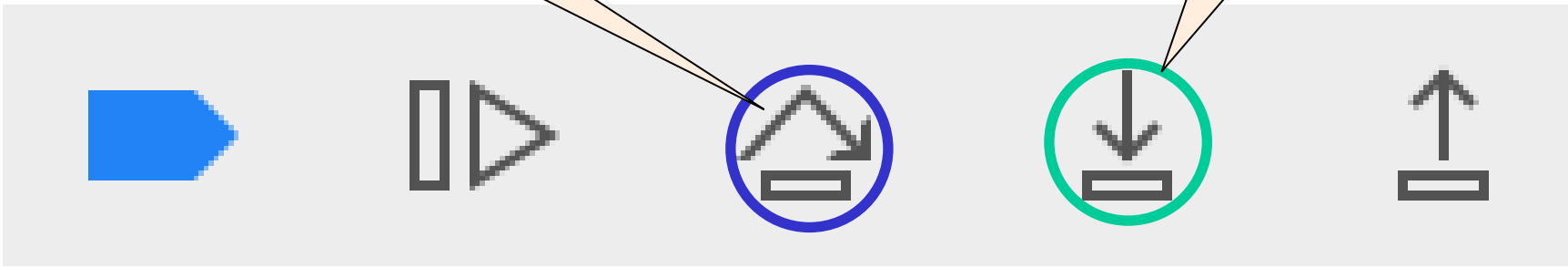
Debugging with Xcode

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

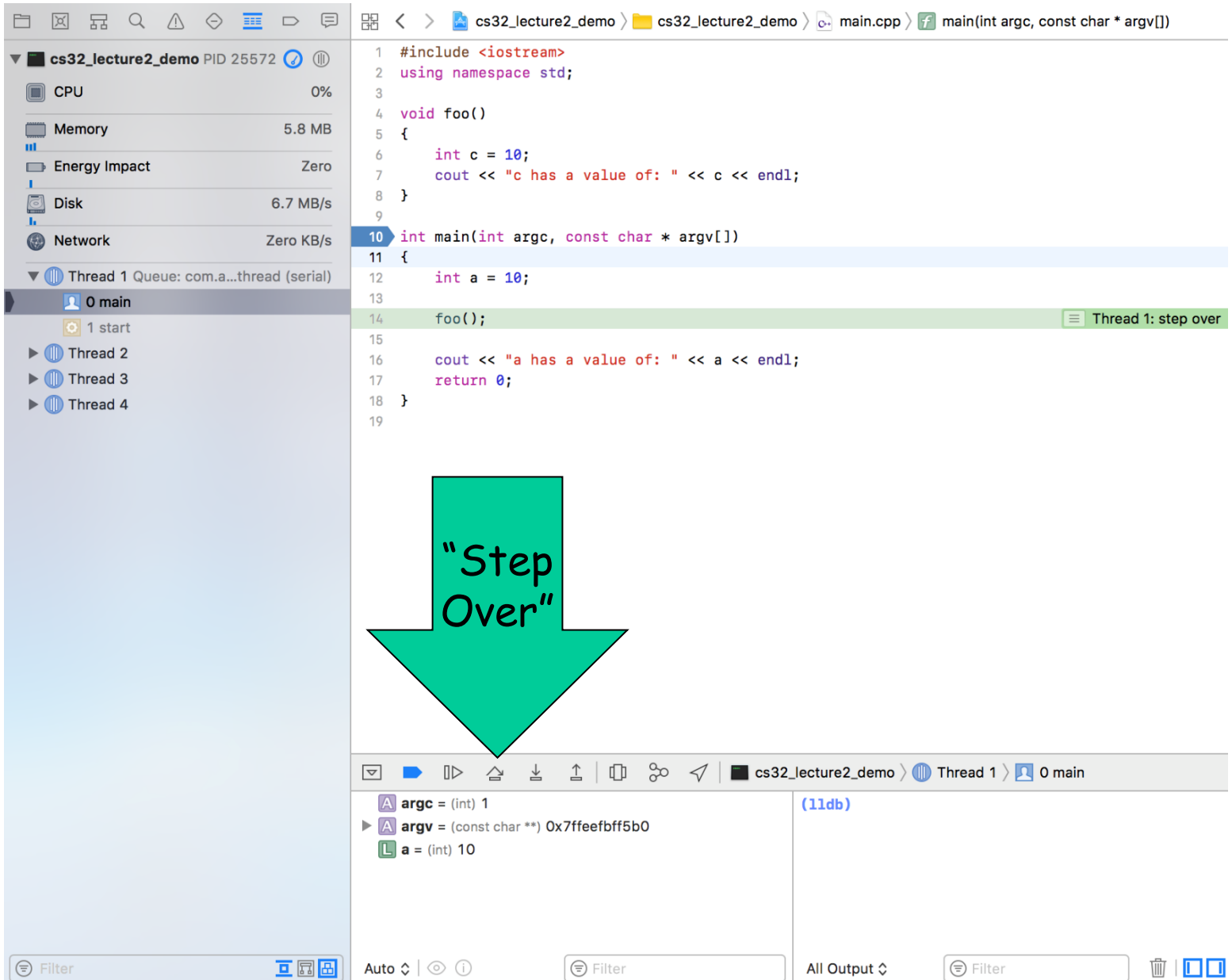
This is the **"Step Over"** button (the button we have been using so far).

The "Step Over" and "Step Into" buttons do slightly different things. Let's see an example.

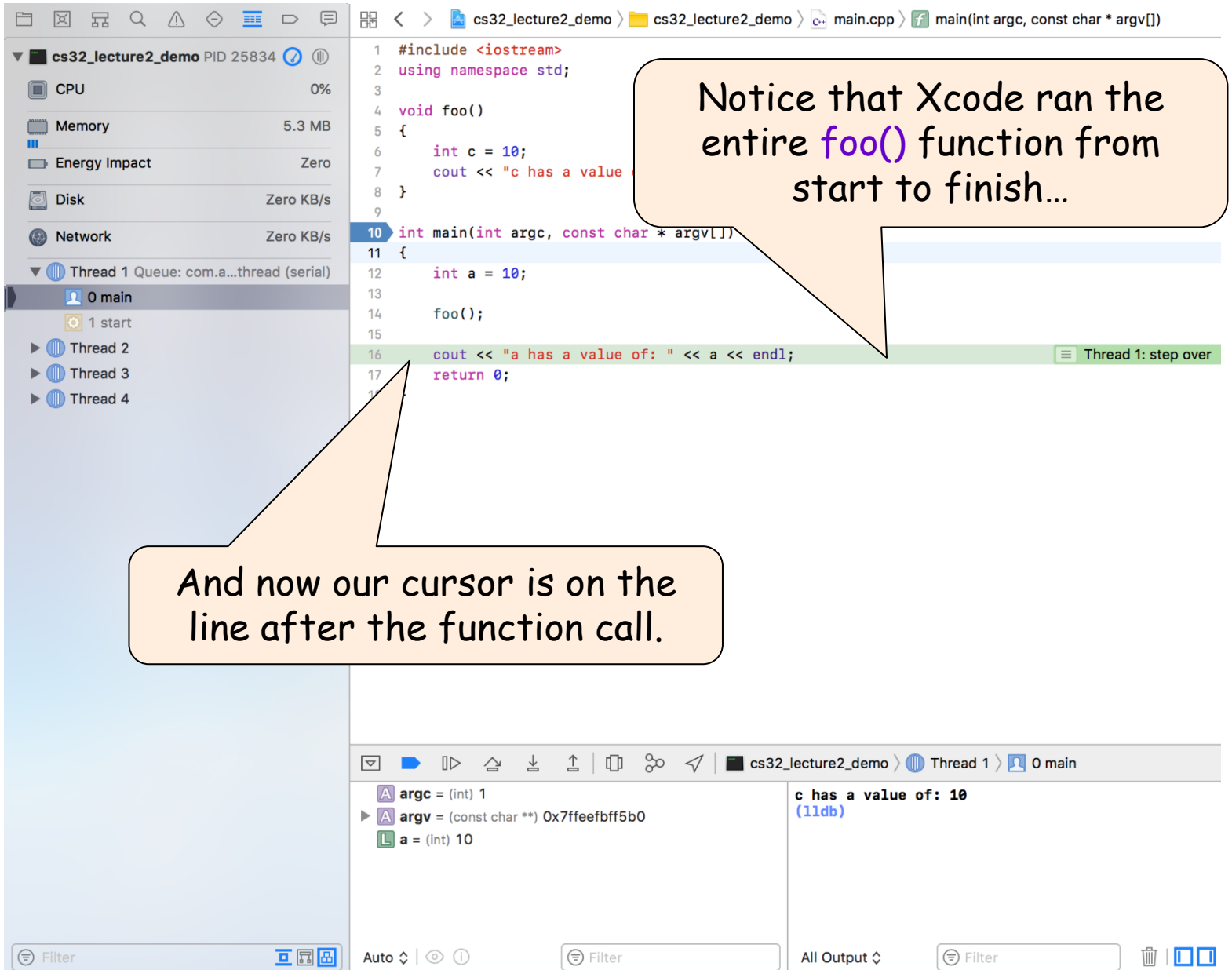
This is the **"Step Into"** button.



Debugging with Xcode



Debugging with Xcode

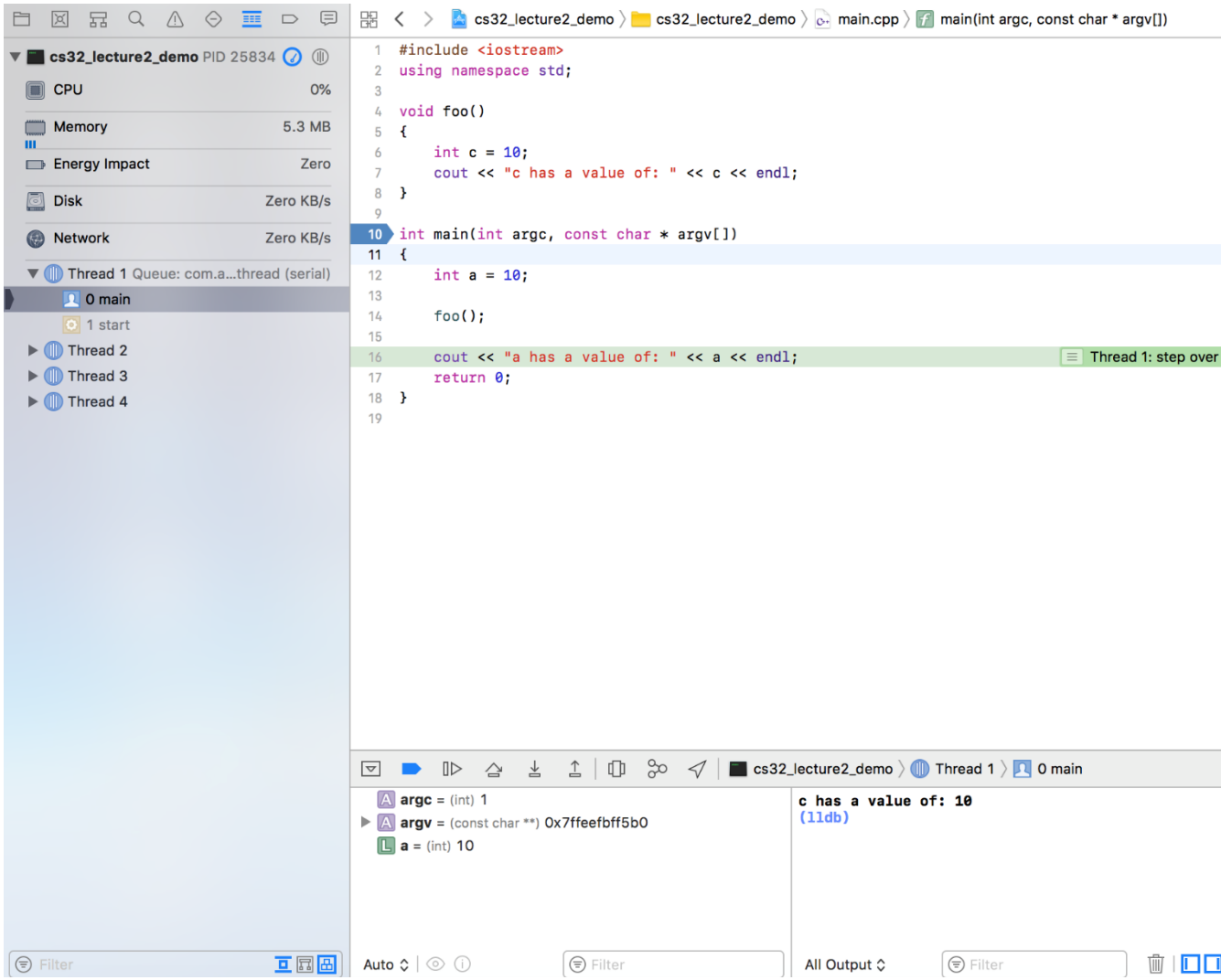


Debugging with Xcode

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.

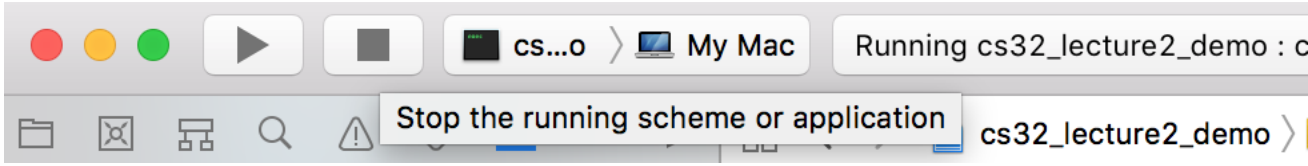
So "Step Over" lets you quickly run a line of code without delving into the details.



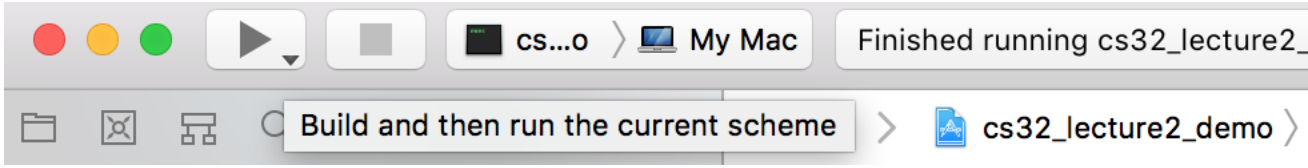
Debugging with Xcode

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

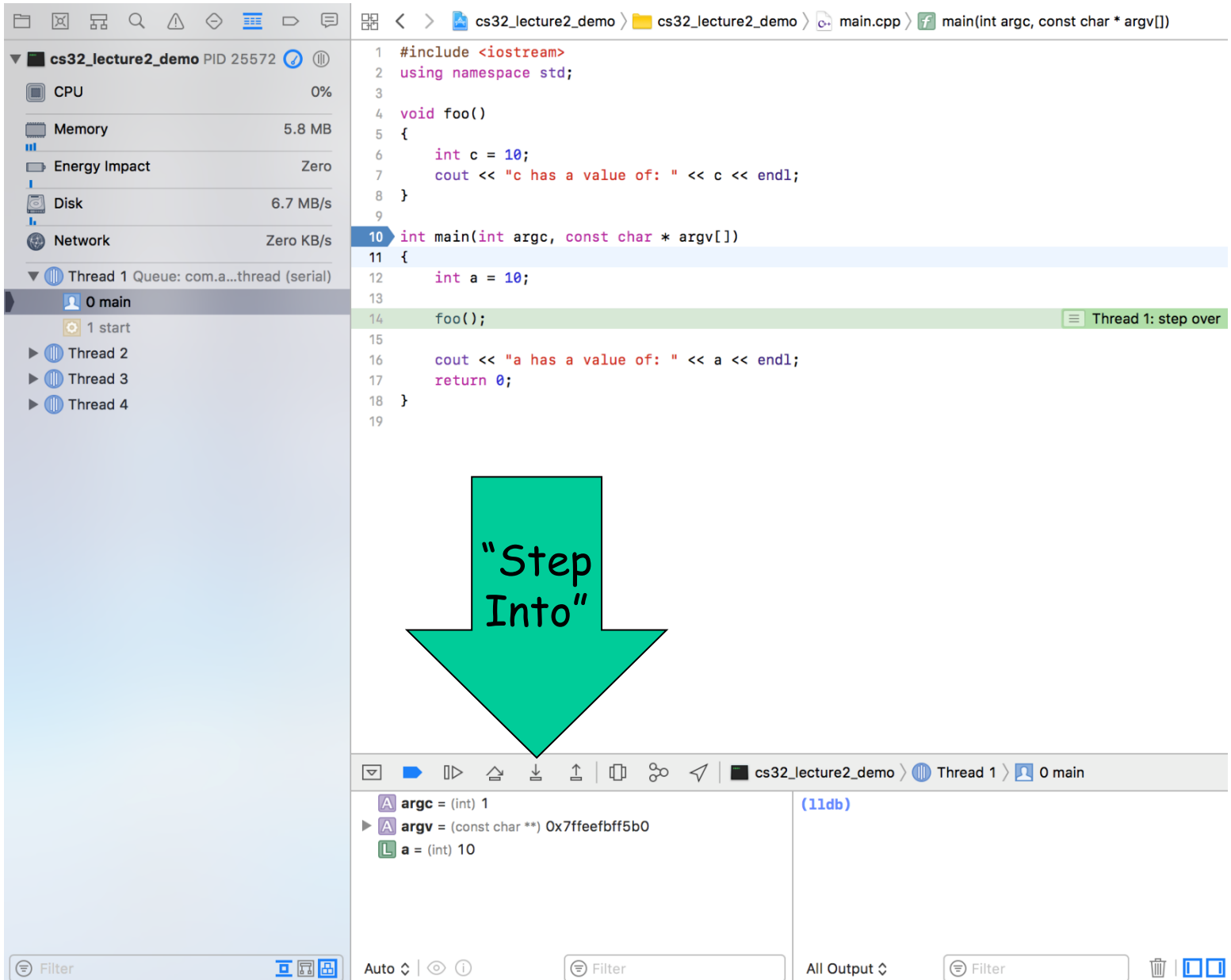
Just tap on the **Stop** button to stop the current debugging session.



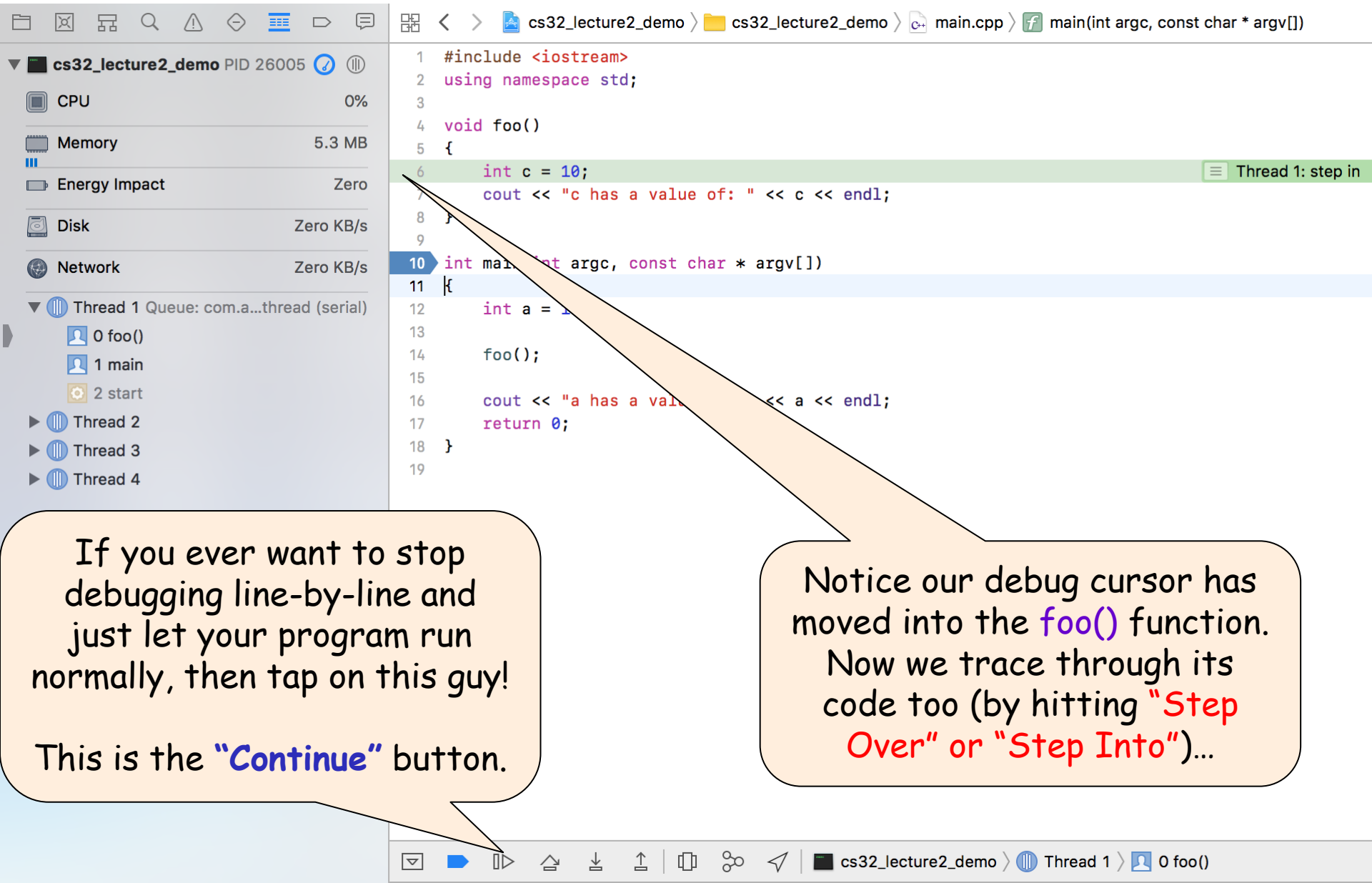
Then tap on the **Run** button to re-build and run the code.



Debugging with Xcode



Debugging with Xcode



If you ever want to stop debugging line-by-line and just let your program run normally, then tap on this guy!

This is the **"Continue"** button.

Notice our debug cursor has moved into the foo() function. Now we trace through its code too (by hitting "Step Over" or "Step Into")...

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?

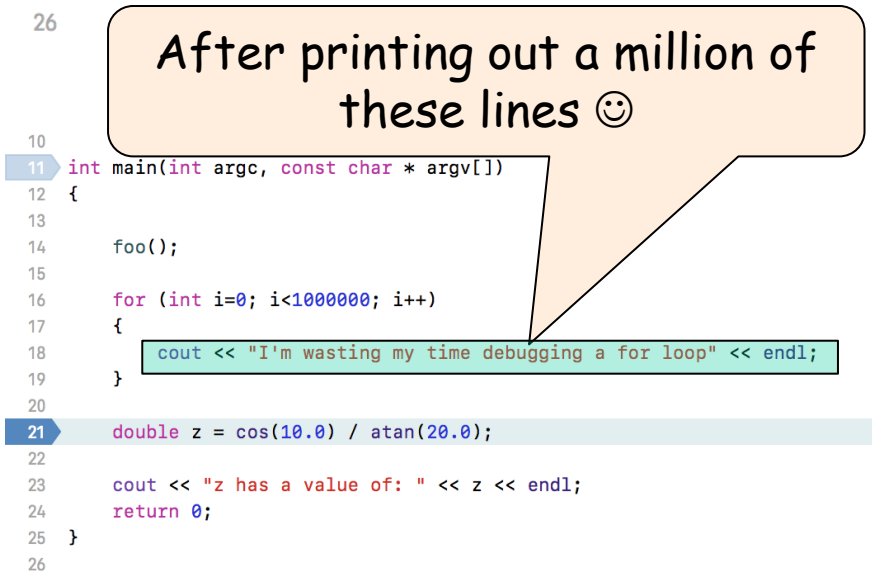
Debugging with Xcode

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];
};
```

main.cpp

```
#include "student.h"

int main()
{
  Student larry;
  Alcohol vodka;

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

- 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!~~

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

```
#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

```
#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;
};
```

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!

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

So I need to include alcohol.h, right?

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:

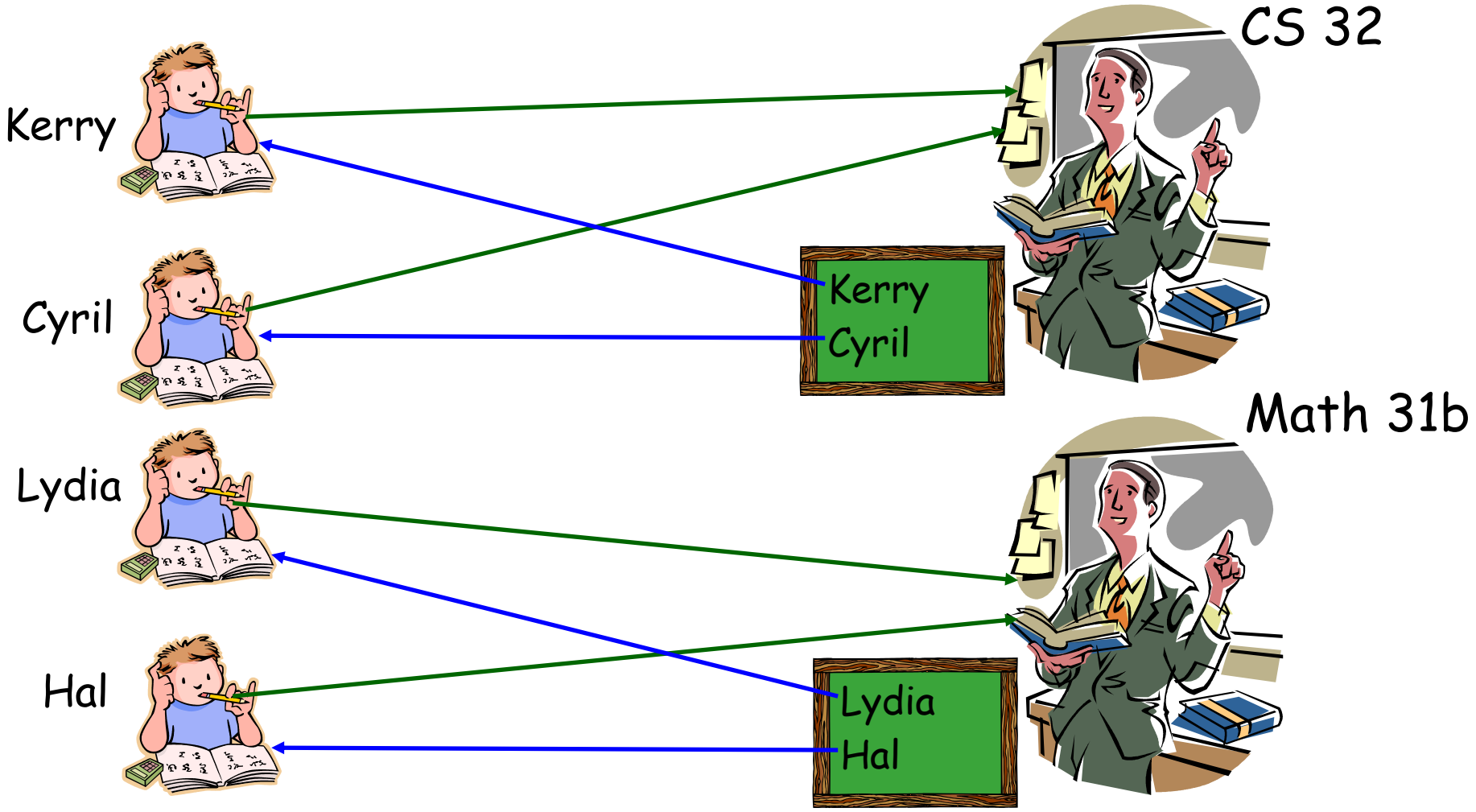
1. If a .h file is large (thousands of lines), #including it when you don't strictly need to slows down your compilation. Especially if you're compiling thousands of files that include your really long .H file.

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)
```

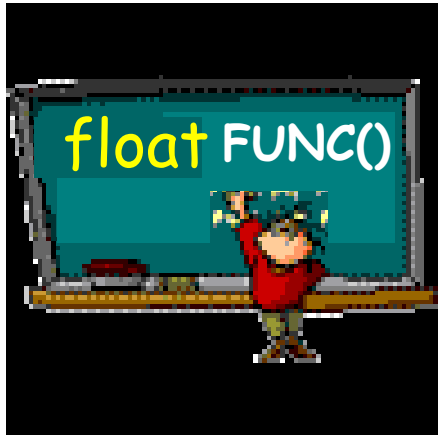
```
// PERFECT! All default parameters come after non-defaults
bool burp(int length, int loudness = 5, int pitch = 60)
{
   ...
}
```

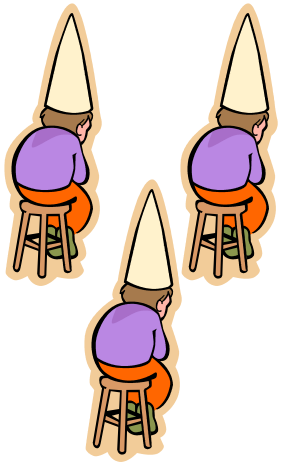
Class Challenge

RULES

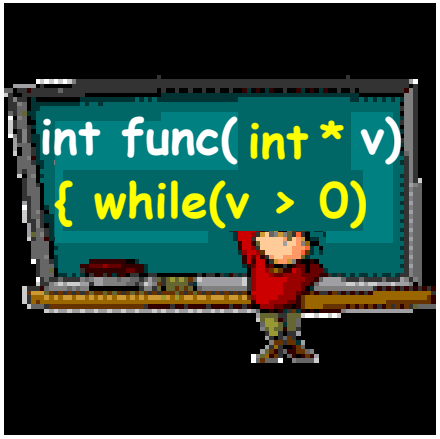
- The class will split into left and right teams
- One student from each team will come up to the board
- Each student can either
  - write one new line of code to solve the problem OR
  - fix a single error in the code their teammates have already written
- Then the next two people come up, etc.
- The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

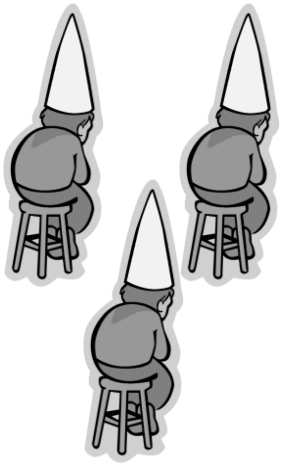
Team #1





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.