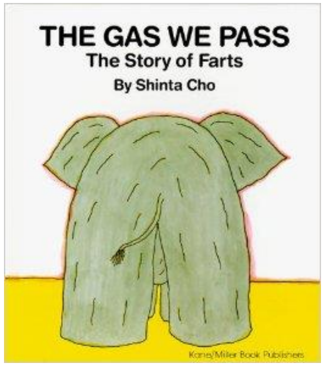
Lecture #2

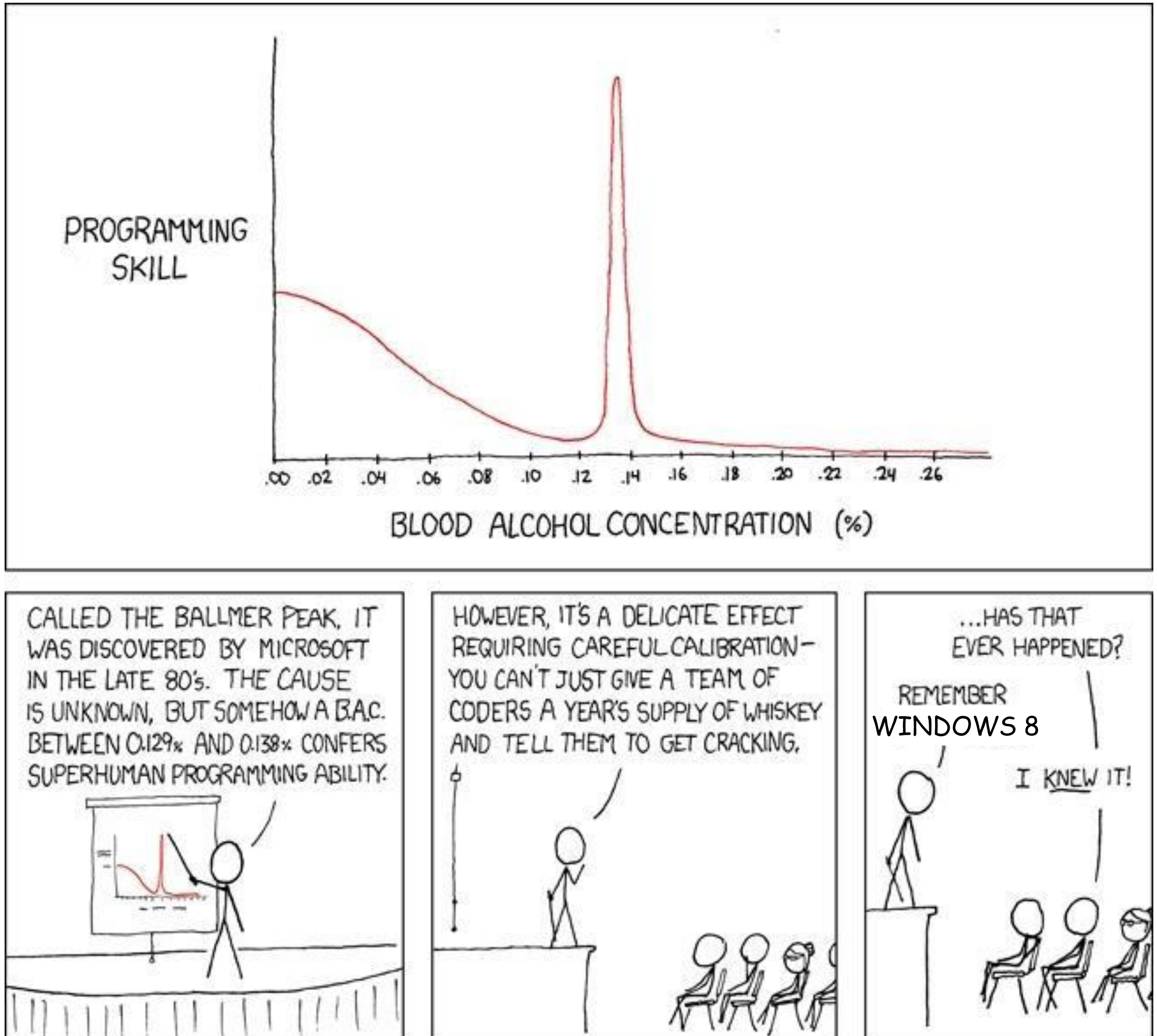
(aka the Fart Lecture)



- 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!


PROGRAMMING SKILL
.00 .02 .04 .06 .08 .10 .12 .14 .16 .18 .20 .22 .24 .26
BLOOD ALCOHOL CONCENTRATION (%)
CALLED THE BALLMER PEAK, IT WAS DISCOVERED BY MICROSOFT IN THE LATE 80's. THE CAUSE IS UNKNOWN, BUT SOMEHOW A B.A.C. BETWEEN 0.129% AND 0.138% CONFERS SUPERHUMAN PROGRAMMING ABILITY.
HOWEVER, IT'S A DELICATE EFFECT REQUIRING CAREFUL CALIBRATION– YOU CAN'T JUST GIVE A TEAM OF CODERS A YEAR'S SUPPLY OF WHISKEY AND TELL THEM TO GET CRACKING.
...HAS THAT EVER HAPPENED?
REMEMBER WINDOWS 8
I KNEW IT!

Constructors, Destructors, etc…

Why should you care?

So we're about to learn a bunch of really boring syntax.

But you can't write C++ programs without this stuff.

And you'll need it for job interviews.

So pay attention!

Constructors: Class Ini

But there's one problem with such an Init function…

What is it?

Every class should have an initialization function that resets new variables before they're used.

```
class Gassy
{
public:
  void Init(int age, bool ateBeans)
  {
    m_age = age; // the person's age
    m_ateBeans = ateBeans;
  }

  int getFartsPerHr(void)
  {
    if(m_ateBeans == true)
      return(100); // explosive gas!
    return(3 * m_age);
```

2 false

david

```
void Init(int age, bool ateBeans)
{
  m_age = age;
  m_ateBeans = ateBeans;
}
int getFartsPerHr(void)
{
   if(m_ateBeans == true)
     return(100);
   return(3 * m_age);
}
 m_age  -32    m_ateBeans  false
```

```
int main()
{
  Gassy  david;
  david.Init(2,false); // old
  cout << david.getFartsPerHr();
}
```

Utoh! Remember, all primitive variables (e.g., ints, bools, etc.) in C++ start out with random values unless they're explicitly initialized!

Right! Our programmer might forget to call the Init function before using their variable…

Constructors

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

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

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

Wouldn't it be great if C++ would guarantee that every time we create a new class variable, it'll be auto-initialized?

Well, as it turns out, that's exactly what the C++ constructor does!

```
int main()
{
  Gassy  david;

  cout << david.getFartsPerHr();
}
```

Since the constructor is called automatically any time you define a new variable… there's no chance of a new variable being uninitialized accidentally.

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

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

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

Instead of being called Init, the constructor function has the same name as the class! (Confusing, huh?)

```
int main()
{
  Gassy  chen(19,true);

  cout <<
   chen.getFartsPerHr();
}
```

hen

```
Gassy(int age, bool ateBeans)
{
  m_age = age;
  m_ateBeans = ateBeans;
}
int getFartsPerHr(void)
{
   if(m_ateBeans == true)
    return(100);
   return(3 * m_age);
}
 m_age  19  m_ateBeans  true
```

The constructor is called automatically every time you create a new instance of your class.

A constructor is a special member function that *automatically* initializes every new variable you create of a class.

Constructors

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

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

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

If a constructor requires parameters:

You must provide values for those parameters when you create a new variable:

```
int main()
{
  Gassy  ed(45,true);// OK

  Gassy  alan; // invalid!
}
```

Constructors

```
class Gassy
{
public:
  Gassy(int age, bool ateBeans)
  {
     m_age = age;
     m_ateBeans = ateBeans;
  }
  Gassy()
  {
    m_age = 1;
    m_ateBeans = false;
  }
  int getFartsPerHr(void)
  {
    if(m_ateBeans == true)
      return(100);
    return(3 * m_age);
  }
private:
  int m_age;
  bool m_ateBeans;
};
```

goes to this constructor, since it has matching parameters.

goes to this constructor, since it requires no parameters.

This variable initialization…

and this initialization…

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.

```
int main()
{
  Gassy  lyn(18,false);
  Gassy  dave;   // OK!!!
}
```

Constructors

```
class Gassy
{
public:
 Gassy() // generated by compiler
 {
   // I don't init primitive member
   // variables… So be careful!!!
 }

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

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

Primitive variables include native C++ types like int, float, double, bool, etc.

Non-primitives include string, Gassy , etc.

These variables will have random values!

If you don't define any constructors at all…

Then C++ generates an implicit, default constructor for you.

Why would it do this? Well, we'll see why in a bit. ☺

But this default constructor won't initialize your object's primitive member variables!

So be careful!

```
int main()
{
 Gassy   carey;

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

When Constructors [partially hidden] Called

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

    Gassy  arr[52];

    Gassy  *ptr = new Gassy(1,3);

    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!

```
                                        #include /*!!TAB=4!*/<stdio.h>
                                        #include /*{{IOCCC2}}*/<SDL.h>
                                        #define b/*{{IOCCC257}}*/ if (
                              #define a(b, c) for (b = 0 ; /*IOCCC/2014*/b<d; b++,c)
                              e,h,f,g,i,j,k,l,m,n,o,p=1,*q,r,s=5, t,*u,x,y,z,A,B, C[
                              333*7],d=333; D,E,F,G [2 ],H, I, J, K, L, M,N= 1,O,P,Q
                              ,*R;char * S, **T;                        /*33*/          ;
                              SDL_Surface*U,* V;                        int*W(          X
                              ){ u=C+X *7; b*u){                        u++; x          =u[0]; y=u[1
                ]; H=u          [2]; z                                  =(H%8)                                 *
                i; A=(          H/8)*i                                  ; B=u[                                 3
                ]/**/;          return                                  u; ; }                                  return
                0; } Y          (){ O=50; r=                                  0; t=1                             ;
                } Z (X          ,m, n,o)/**/                                  { /**/                             ;
                return          W(X)&&B&&(m<                                  x+i&&n                             <
                y+i&&x<m+i&&                                            y<n+o); } int*ba(int*u){ int X
                ,*bb=C; a(X,                                            bb+=7){ b!*bb){ *bb=1; R
                =bb +1; H=6;                                            while(H--)*R++=*u++; /*W
                                       N                                                                                 N
                                       E                                                                         E
                                       S                                                                                 S
                              */return bb; } } return 0; } bc(e){ q[2]=e; } bd
                              (be,bf){ int X,bg; m+=be; n+=bf; I=e
                              -i; m=m<0?0:(m>I?I:m); a(X,0){ b Z(X
                              ,m,n,o)){ bg=B&1; b D&&bg) continue;
                m-=be; n-=bf; b B&8){ u[-1]=0; j=1; bc(8); b B&32){ n-= i;
                o=i*2; } } b B&16&&!O){ bc(32+(o>i?8:0)); Y(); u[- 1]=0; }
                b(B&128&&bf&&s<0)||(B&64))u[-1]=0; b bg&&!O){ b bf&&s >0){
      u[2]--; u[3]=bf=0; s=-6; } else { b j){ bc(0); b o>i) n+=o-i; D=30; j=0;
      } else { bc(24); Y(); L=-1; } } } b B&4){ b bf&&s<0){ int I[]={ x,y-i,u[
      5],u[4],rand()%2?1:-1,2} ; u[2]++; u[3]=2; ba(I); } } b bf)s=1; break; }
      }                       } bh (m,e,k,     bi){ H=k/ 2;           G[bi]=m>e-H?               k
      -                       e:(m>H?H-m:0     ); } bj(X,be           ,bf){ int bk               ;
      u                       [0]+=be; u[1     ]+=bf; W(X);           E=x,F=y,I=0;               a
      (                                 bk,0){ b I=(X!=bk&&Z(bk,E,F,i)&&(B&6                             )
      )                                 )break; } W(X); b I){ b bf)u[1]-=bf;                             b
      be                        ){ u[0]-=be; u[4]*=-1*be; } } W( X);                             }
      bl                      (){ int bm=n,X; SDL_FillRect(U,0,M); X=4; while(                   X
      --                      )bd(r,0); X=3; while(X--)bd(0,s); b n>h&&!O)Y ()                   ;
      t                       =bm=-n; q[0]=m; q[1]=n; bh(m,e,k,0); bh(n,h,l,1)                   ;
      a(X,0){ b W(X)){ b B&9){ bj(X,u[4],0); bj(X,0,2); } b B&1){ *u +=u[4]; b
                              ++u[5]>20){ u[4]*=                        -1; u[5]=0; } z+=K
                              %2?i:0; } J=i; b q                        ==u){ J=o; b!O){ b
                              r)z+=i*(K%2); b!t)                        z=48; z+=p<0?i*4:0
                              ; }} b q!=u||!(D&&                        0==D%3)){ SDL_Rect
                bn={ z,A,i,J} ,bo=                                            { G[0]+x,G[1]+y,i,
                J}; SDL_BlitSurface                                           (V,&bn,U,&bo); } }
                } b(s+=2)>2)s=2; K                                            ++; D-=D?1:0; b O)
                b!-- O) exit(L); }                                            bp(X){ return strtol
      (T[X],0,0); } bq(int H ,                                                int*br,int bs ){ int bt;
      bt=Q-P; b bt>bs) bt= bs;                                                SDL_MixAudio(br,S +P,bt,
      128); P+=bt; b P>=Q)P=0;                                                } SDL_AudioSpec bu={8000
      ,8,1,0,256,0,0,bq} ,bv ;                                                main(int bw,char**bx){ T

=bx; int by,H=255,   bz=H<<8                                  ,bA =bz   <<8,bB= bA<<8,
bC=bp(5),bD=bp(6);   SDL_Event                          bE; o =i=  bC / 8; k = bp(1)
,l=bp(2),e=bp(3),h   =bp (4); M=                      bp(9); /**/  SDL_Init(/*<2053"*/
0xffff                           ); b!*(char*)          &N){ H=bB; bz   =bA; bA         =bz>>8;
bB=255                           ; } U=SDL_SetVideoMode(k,l,0,0);  V=/*NES       HISTORY
SANDRO                           */ SDL_CreateRGBSurface(1 <<15 ,  bC, bD,       32,H,bz
,bA,bB); fread(V->   pixels,bC*bD*4,1,fopen(T [7],"r"   )); SDL_OpenAudio(&
bu,0); SDL_LoadWAV   (T[8],&     bv,&S,&Q             ); /**/   SDL_PauseAudio(0);
for(; ; ){ int u[6   ],*I; H             =0           ; while   (H < 6) scanf(/*0*/
                        "%d ",   u+H++);                                      b u[5]<   0)break          ; I=ba(
                        u);b!u   [3]){ q                                      =I+1; m   =u[0 ];          n=u[1];
                       } } for   (; ; ){                                      while/*   !MAFFIO          DO.S*/(
SDL_PollEvent(&bE)   ) { by=                                  bE.type   ==3; b!O &&(by|| bE.
type==2)){ I = bE.   key/**/                                  .keysym   .sym; b I== 276)r=
by?0:(p=-1); b I==   275 )r=                                  by?0:(p   =1); b I==32)by?

0:(t?(s=-9):0); b I==27)exit(0); } } bl(); SDL_Flip(U); SDL_Delay(60); } }
```

Destructors

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 <u>destruct</u> a class variable when it goes away.

Here's how we define a destructor directly inside our class definition…

```
class SomeClass
{
public:

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

private:
   …
};
```

Destructors must NOT have any parameters.

It looks just like a constructor function except for the tilde ~ which identifies it as a destructor.

To define a destructor function, place a tilde ~ character in front of the name of the class.

Destructors must NOT return a value either.

Destructors

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

  int getNerdScore(void)
  {
    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.

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.
      downloadMP3ToReservedSp
      // Then play the song that
      playMP3InReservedSpace();
   }

};
```

We reserve 100MB of disk space when we construct a new Spotify object.

But nothing in our class ever frees this disk space!

So every time we create a new object, we waste another 100MB!

```
int main()
{
   // Obsessively play EDM music!
   for (int i = 0; i < 100; ++i)
   {
     SpotifyPlayer a;
     a.playSong("http://spotify.com/edm.mp3");
   }
}
```

+300MB

+200MB

+100MB

Why Do We Need Destructors?

```
class SpotifyPlayer
{
public:
   SpotifyPlayer()
   {
      // Reserve 100MB of      space to temporarily
      // cache downl    aded MP3 music files.
      reserveSpaceOnDisk(100000000);
   }
   ~SpotifyPlayer()
   {
      // Free the r        ed space on disk.
      freeReservedSpace();
   }
   void playSong(string songURL)
   {
      // First download song to r
      downloadMP3ToReservedSp
      // Then play the song that
      playMP3InReservedSpace();
   }
};
```

Now every time we construct we reserve 100MB…

And every time we destruct we free our 100MB!

Let's add a destructor, of course!

```
int main()
{
   // Obsessively play EDM music!
   for (int i = 0; i < 100; ++i)
   {
      SpotifyPlayer a;
      a.playSong("http://spotify.com/edm.mp3");
   }
}
```

+100MB

Can anyone see what the problem is?

So how do we fix our code???

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 a disk file	Closes the disk file
Connects to another computer over the network	Disconnects from the other computer

Don't forget or you'll FAIL!

Destr

So when is a destructor called anyway?

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


  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
```

Local objects defined in a function are destructed when they "go out of scope."

Local variables defined in a block are destructed when the block finishes.

Dynamically allocated variables (e.g., allocated using the new command) are destructed when delete is called, *before* the memory is actually freed by the OS.

Destructors

So when is a destructor called anyway?

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

```
void Dingleberry(void)
{
    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;


}
```

The destructor is called 42 times for x here!

Class Challenge

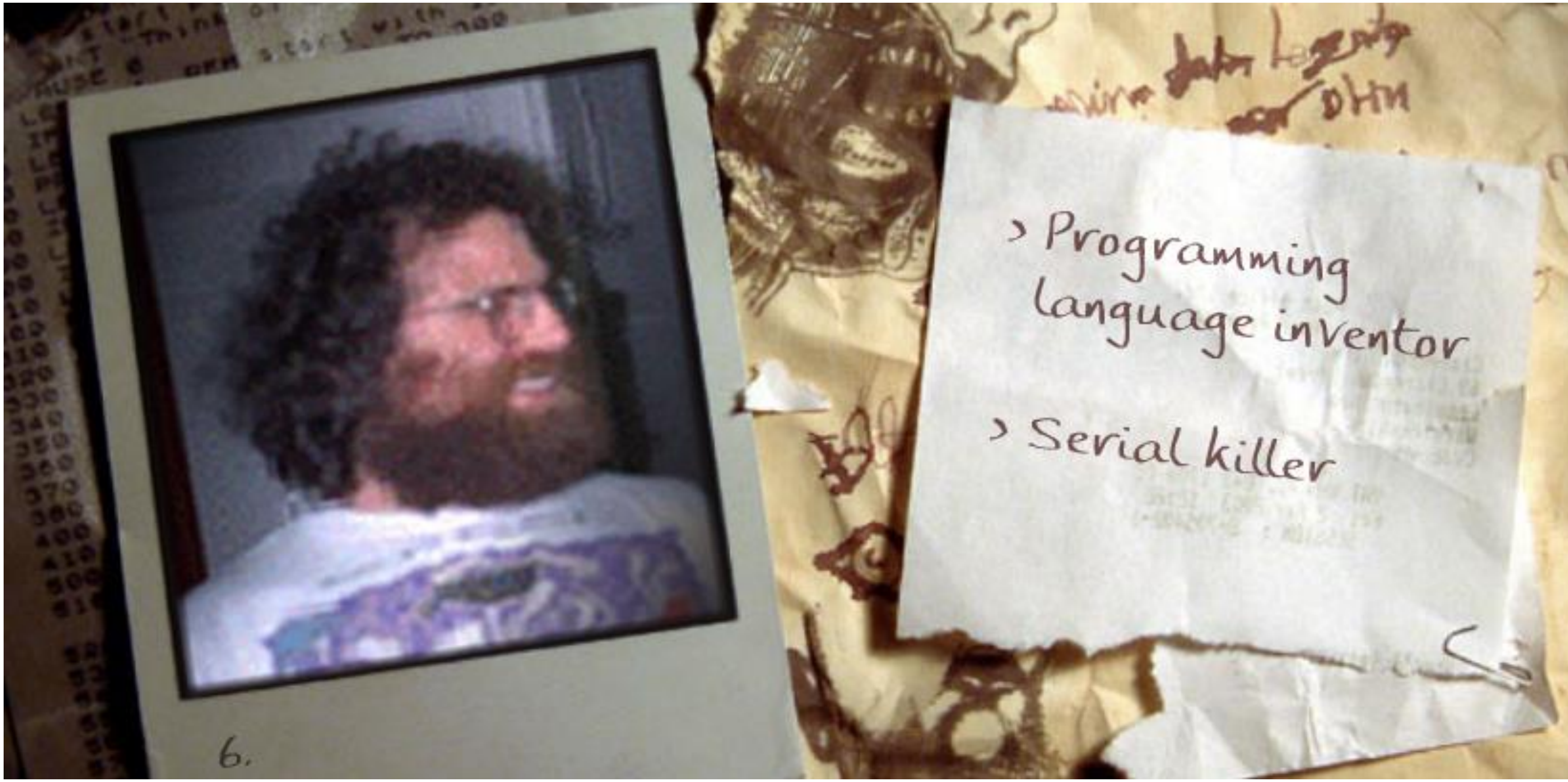
Name all the times Gassy's c'tor and d'tor are called in this example…

```
class Gassy
{
public:
 Gassy()
 {
   m_age = 1;
   m_ateBeans = true;
 }

 ~Gassy()
 {
   for (int i=0;
        i<getFartsPerHr();i++)
      cout <<"Poof!";
 }
 ...
private:
  int m_age;
  bool m_ateBeans;
};
```

```
void foo()
{
   Gassy a, *b;
}

int
{
   int j;
   Gassy *ptr, x;

   ptr = new Gassy
   for (j=0;j<2;j+
   {
       Gassy a[500000];
       foo();
   }
   delete ptr;
}
```

Question: How could we make our program more efficient? (Hint: Look where this is pointing)

Can you guess?



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

Class Composition

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

Note: string is a class just like Car and GasTank! So if your class contains a string member, you're doing composition!

```
class GasTank
{
  ...
};

class Car
{
  ...
private:
  GasTank myTank;
  int horsePower;
  string modelName;
};
```

myTank is an object of the GasTank class.

So our Car class is using class composition.

Class Composition

Class Composition

Class Composition

Class Composition

```
class Valve
{
  ...
};

class Heart
{
  ...
 private:
   Valve m_valves[4];
};

class TinMan
{
  ...
private:
 Heart m_myHeart;
};
```

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;
};
```

So how does construction work when we use class composition?

Well, let's assume that we have three classes...

So if I define a HungryNerd variable, how is it actually constructed?

Well, a HungryNerd has a constructor. But it also contains a Stomach and a Brain, and they have constructors too!

So how does C++ handle all this?

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

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;
};
```

So how does construction work when we use class composition?

Well, let's assume that we have three classes...

So if I define a HungryNerd variable, how is it actually constructed?

Well, a HungryNerd has a constructor. But it also contains a Stomach and a Brain, and they have constructors too!

So how does C++ handle all this?

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

```
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;
};
```

Uses myBelly and myBrain!

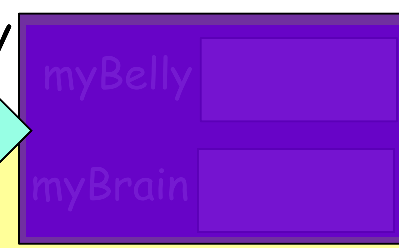
Well, HungryNerd's constructor should run first, right?

But wait! Our HungryNerd constructor uses its Brain and Stomach variables…

But how can it use these variables if they haven't yet been constructed?

It can't! HungryNerd needs to construct these variables first, before its own constructor can run and use them!

carey



But they're not initialized yet!

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

```
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

And that's exactly what happens!

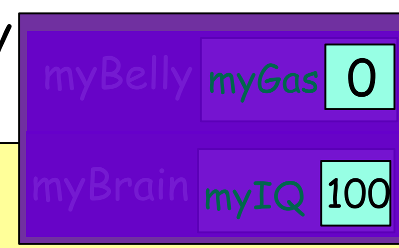
When an outer object contains member objects…

C++ automatically adds code to the outer object's constructor to FIRST call the DEFAULT constructors of all the member objects…

And then AFTER they're constructed, it runs the body of the outer object's constructor.

Now our outer constructor can safely use any of the member variables!

carey




```
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
   }
priv
   St
   Bra
};
```

Uses myBelly and myBrain!

So they can only be destructed after the outer destructor finishes!

OK, that makes sense.
But what about destruction?

At the time when our object (carey) goes out of scope, C++ automatically calls its destructor.

OK, but which destructor runs first? HungryNerd's, Stomach's or Brain's?

Well HungryNerd is allowed to use its member variables in its destructor…

So they can't be destructed until HungryNerd's destructor finishes!

carey

myBelly	myGas	1
myBrain	myIQ	110

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

C++ calls carey's destructor

```
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 myBelly 's destructor
     Call myBrain 's destructor
private:
   Stomach myBelly;
   Brain       myBrain;
};
```

So what happens?

The C++ compiler adds code to the end of the outer destructor to call the inner objects' destructors.

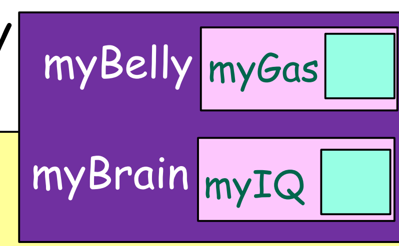
So the outer destructor runs first.

Then, after the outer destructor runs, it calls the destructors of the inner objects (in reverse order of construction).

Finally, after all embedded objects have been destructed, the outer object's memory is freed.

And of course, if an embedded object has its own embedded object(s), this process is repeated.

carey




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

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

So why does C++ auto-generate constructors and destructors for your classes if you don't define your own?

Answer:
To ensure that each member object is properly constructed and destructed!

This ensures that by the time your other member functions use your embedded object variables, they'll be properly initialized!

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
  }

  void celebrate()
  {
     cout << "Happy " << myAge <<
          "'th birthday!";
  }
private:
     Belly myBelly;
     Brain myBrain;
     int   myAge;
```

This implicitly added constructor…

so they'll have a random value here!

wont initialize your object's primitive variables…

Just be careful!

Remember, that the auto-added default constructor…

WON'T initialize your primitive member variables!

So if your class has any primitives (ints, bools, floats, doubles, etc.) you should define your own constructor and initialize them!

Class Composition Construction + Destruction

Alright, so here's the order the constructors will run…

And here's the order the destructors will run…

Since Ack has no embedded variables, the preamble does nothing.

```
class Ack
{
public:
                         #4
    Ack()     N/A        { … }
    ~Ack()               { … }   N/A
};                       #6
```

```
class Bar
{
               #3        #5
public:
    Bar()     Call Ack() { … }
    ~Bar()               { … }   Call ~Ack()
private:                 #4      #5
    Ack       myAck;
};
```

```
class Foo
{
               #2        #6
public:
    Foo()     Call Bar() { … }
    ~Foo()               { … }   Call ~Bar()
private:                 #2      #3
    Bar       myBar;
};
```

Since Ack has no embedded variables, the postamble does nothing.

```
int main()
{
#1 Foo      x;
    …
} #1
```

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... si...

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

And here's how we create a new Stomach variable…

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

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

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

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

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!

Advanc... mposition

Because our new Stomach class REQUIRES you to pass a value in to its constructor.

```
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;
```

But our Stomach class doesn't have a default (parameter-less) constructor anymore!

But our auto-generated C++ code always calls the default constructor, which takes NO parameters!

So this results in a C++ compilation error!

Advanced Composition

And pass in a value so it can be properly constructed!

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

Well, our HungryNerd constructor needs to somehow explicitly call myBelly's constructor...

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

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

Ok, so how can we fix our HungryNerd class so it works with our new Stomach class?

A...d Class Composition

whose constructor requires parameters!

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

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

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

This is called an "initializer list".

Instead, we'll add our own C++ code to explicitly call myBelly's constructor with a parameter.

We use it to initialize an embedded object…

Alright, let's see the "exciting" C++ syntax!

Advanced Class Composit

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

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

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

You must add an initializer list to *all* of your outer class's constructor(s).

The initializer list sits between the constructor's prototype and its body.

It starts with a colon, followed by one or more member variables and their parameters in parentheses.

Any time you have a member variable (e.g., myBelly) that requires a parameter for construction…

So here's what your revised C++ code looks like (without the C++ magic).

Advanced Class Compos

```
class Stomach
{
pu

pr

};
```

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

};
```

```
class HungryN
{
public:
   HungryNerd()
     : myBelly(10)

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

private
   Stom
   Brain
};
```

This line calls myBelly's constructor with a parameter value of 10!

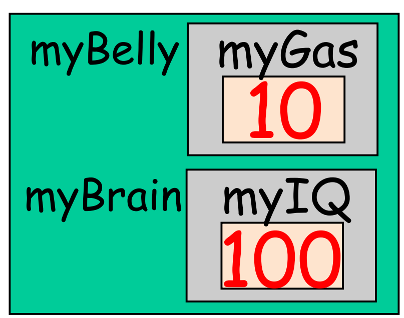
Call myBrain's default constructor

Of course, C++ is still happy to implicitly construct any embedded objects with default constructors for us!

Finally, once all of our embedded objects have been constructed, C++ will run the outer constructor's body!

OK – let's see our new HungryNerd and Stomach classes in action!

carey



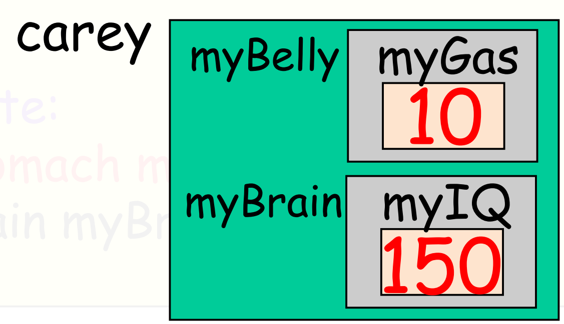

```
int main(void)
{
   HungryNerd   carey;

   ...
}
```

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(int startIQ)
    {
       myIQ = startIQ;
    }
    void think() { myIQ += 10; }
};
```

```
class HungryNerd
{
public:
   HungryNerd()
     : myBelly(10), myBrain(150)
   {
      myBelly.eat();
      myBrain.think();
   }
```




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

    ...
}
```

Initializer Lists

If you like, you may also use the initializer list
to initialize your primitive member variables too.

```
class HungryNerd
{
public:

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

 }

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

You may do it here in your initializer list…

Feel free to use the initializer list for primitives if you like, but it's not required either way.

This will initialize your primitive variable(s) before the body of your constructor runs.

Instead of initializing your primitive variables here in your constructor body…

Advanced Class Compo

We don't need to have constants here!

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

```
class HungryN
{
public:
   HungryNerd(int startingGas)
     : myBelly( startingGas )
   {
     myBelly.eat
     myBrain.th
   }

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

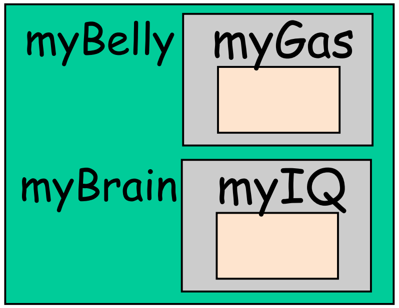
Call myBrain's d   fault constructor

Now we'll initialize our myBelly member with whatever the user passes in to the HungryNerd constructor!

And these can even be more interesting expressions!

Finally, there's no reason why the parameters to your initializer list have to be constants…

carey




```
int main(void)
{
    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

Why should you care?

You're lazy, right?

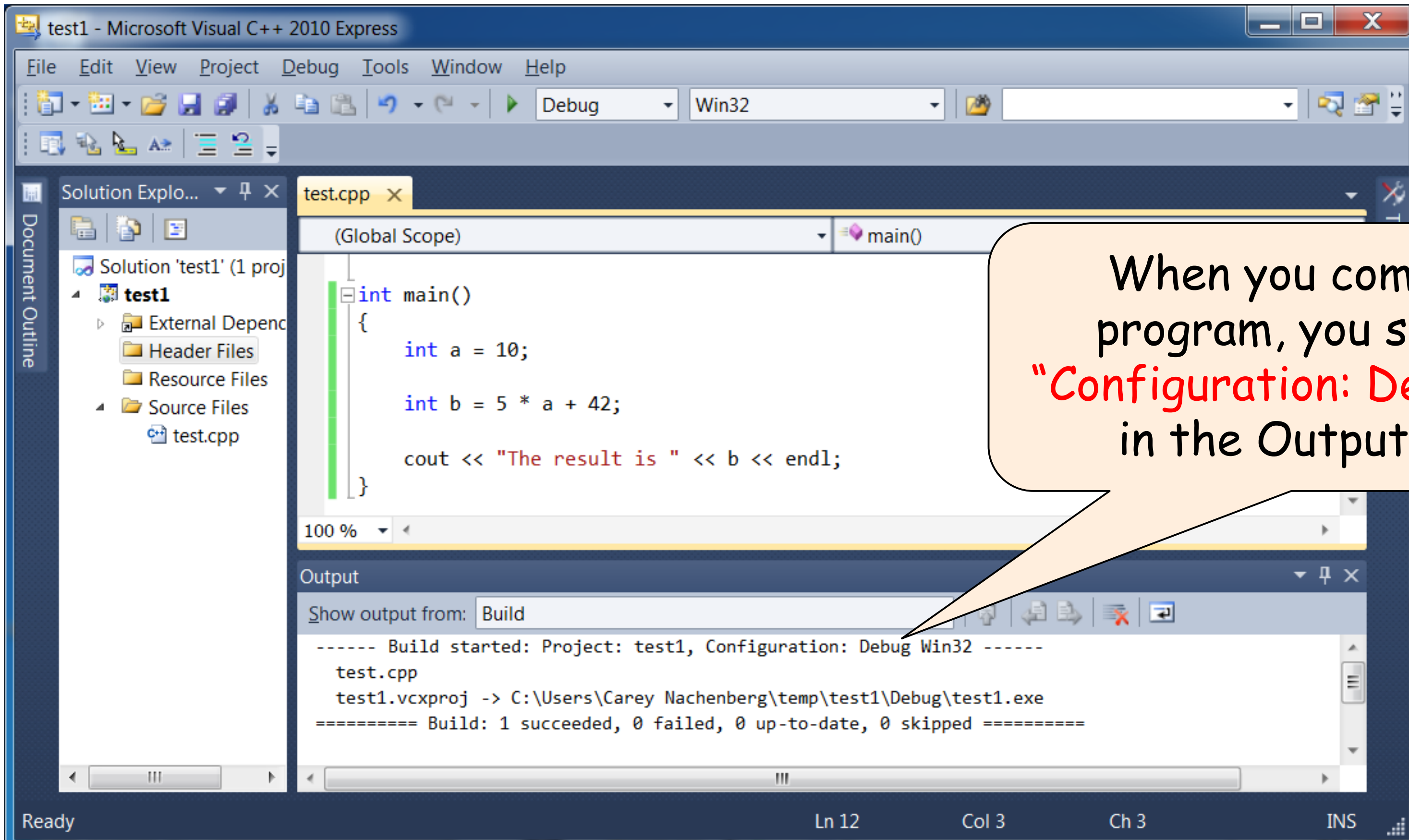
You'd rather not waste 2 hours finding a bug...

When you could find it in 30 seconds with the debugger.

So pay attention!

Debugging

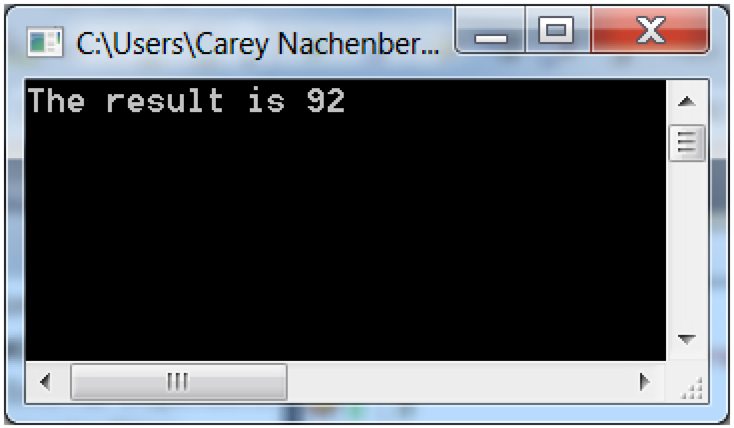
By default, Visual Studio compiles all programs in "debug mode" so you can use the debugger with them right away.



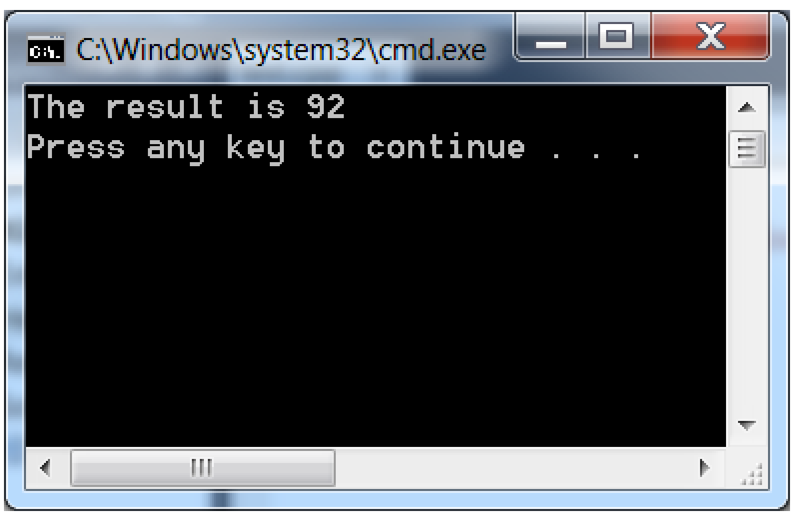
When you compile your program, you should see "Configuration: Debug Win32" in the Output window.

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.

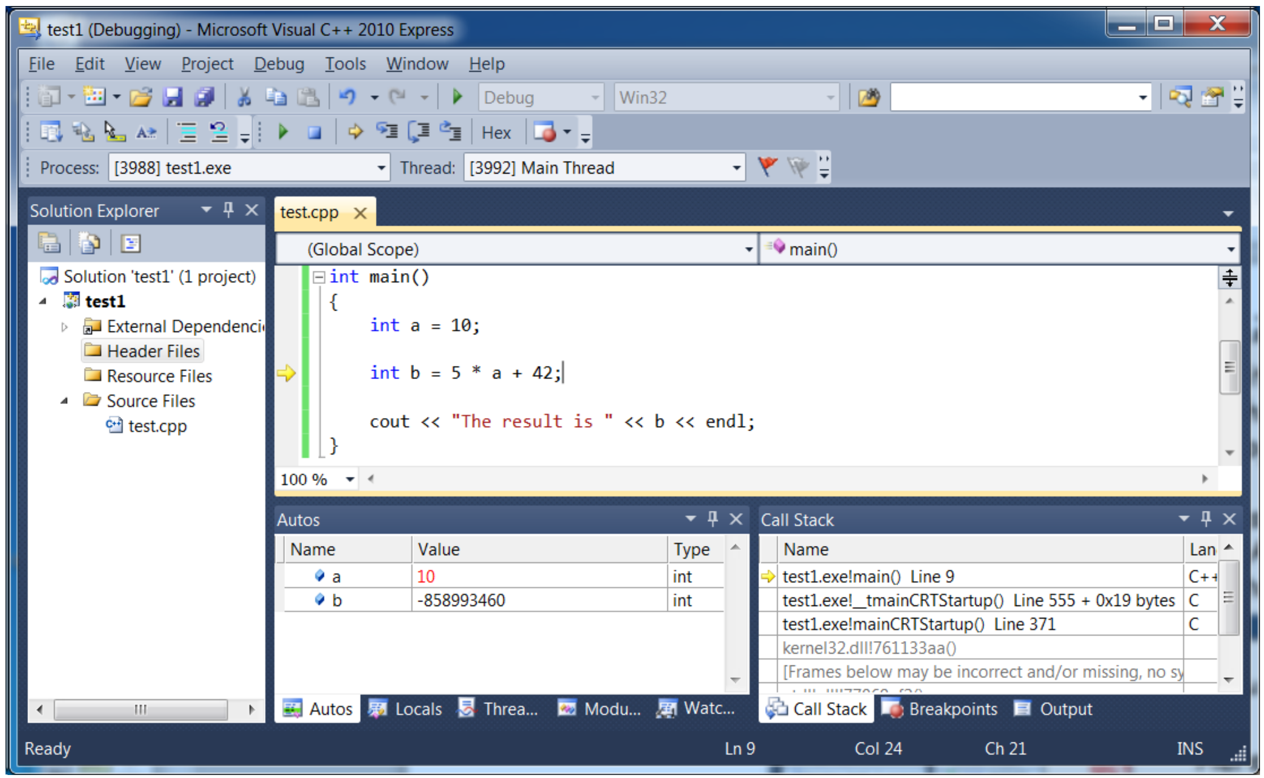


If you run by hitting Ctrl-F5, Visual Studio will print "Press any key to continue…" and leave the debug window on the screen when your program completes.

Debugging

You can step through your program one line at a time using the F10 and F11 keys.

To start your program and debug one line at a time, hit F10 after you finish compiling it.



To run the current line of code, hit F10 or F11.

Let's do that now.

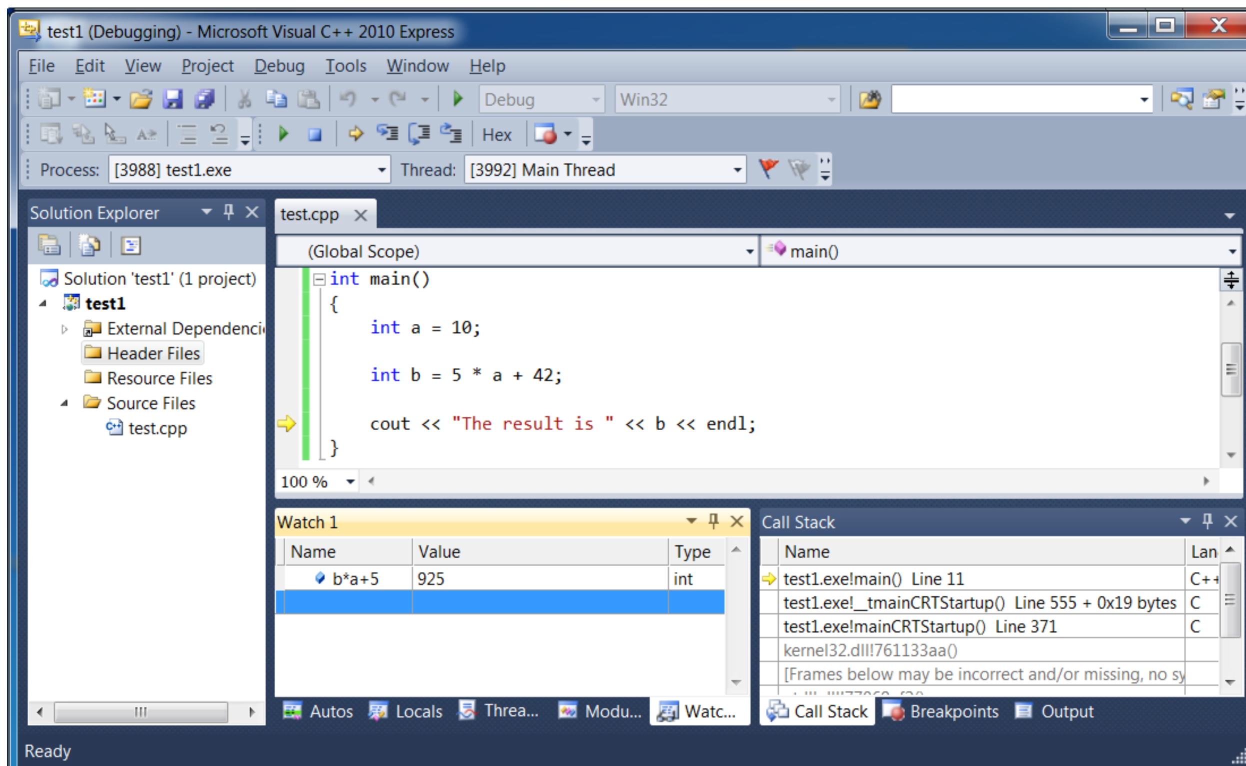
Let's hit F10 again and trace over another line.

Debugging

You can step through your program one line at a time using the F10 and F11 keys.

Let's hit F10 again and trace over another line.

If you want to watch other variables or expressions, you can do that in the "watch" sub-window.

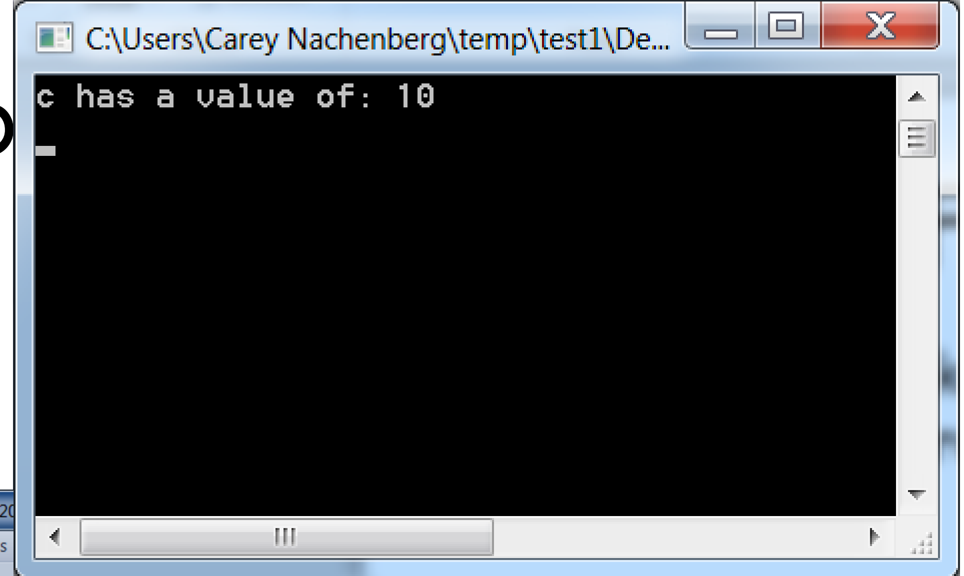


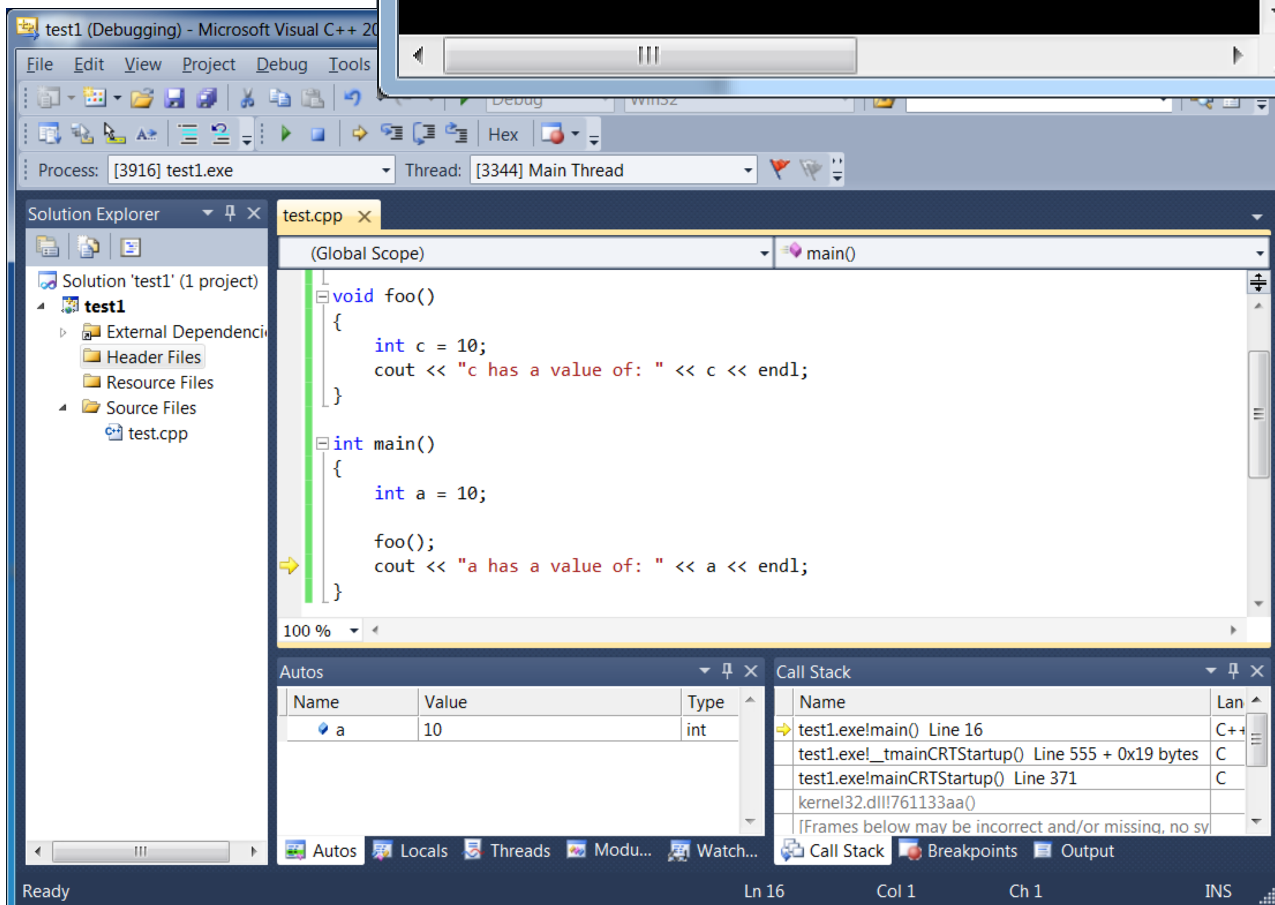
Deb

The F10 and F11 keys actually do slightly different things…

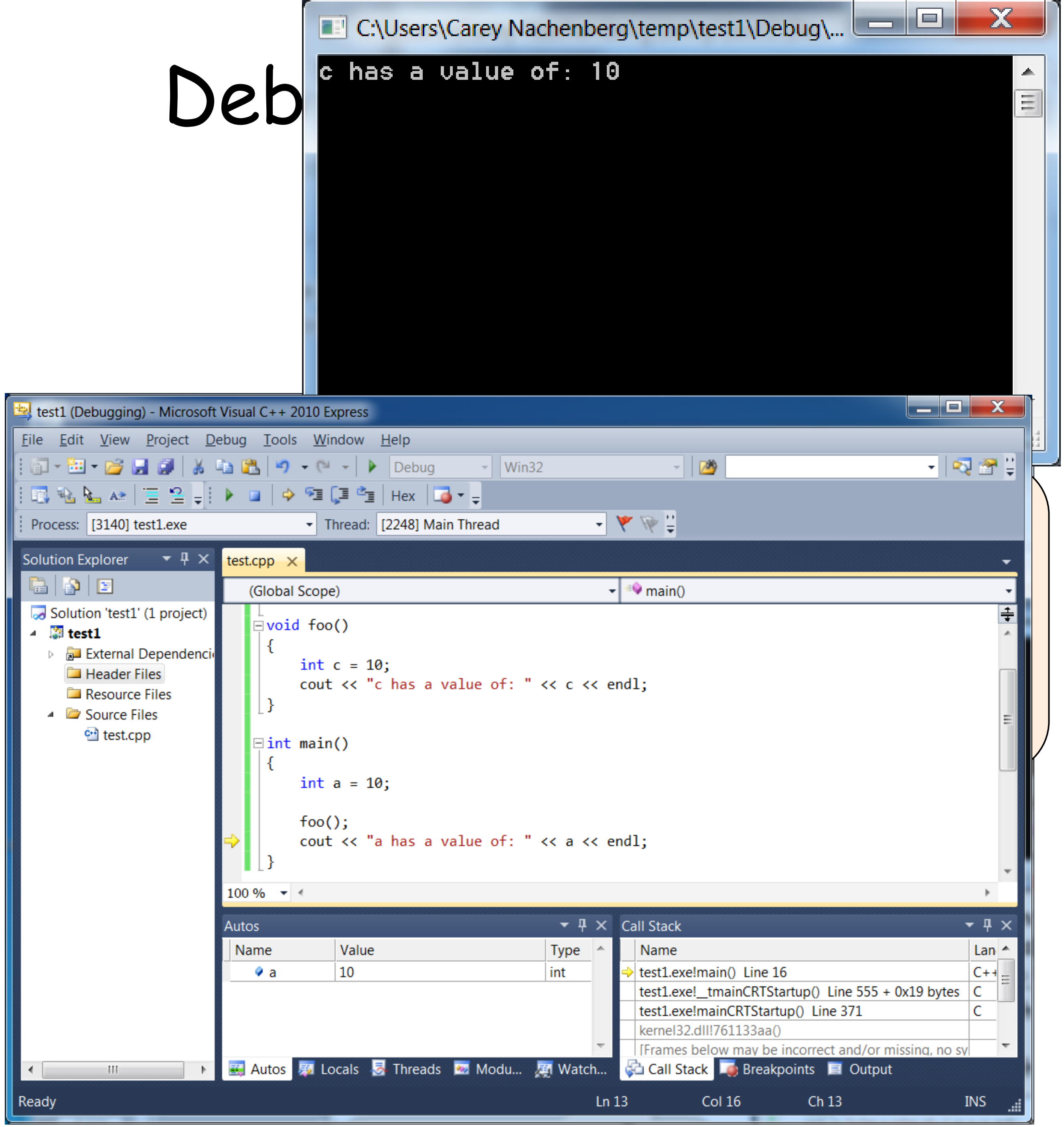
If you hit F10 while on a function call line, Visual Studio will run the entire function in one step. You won't be able to trace line-by-line through it.

So F10 lets you quickly run a line of code without delving into the details.





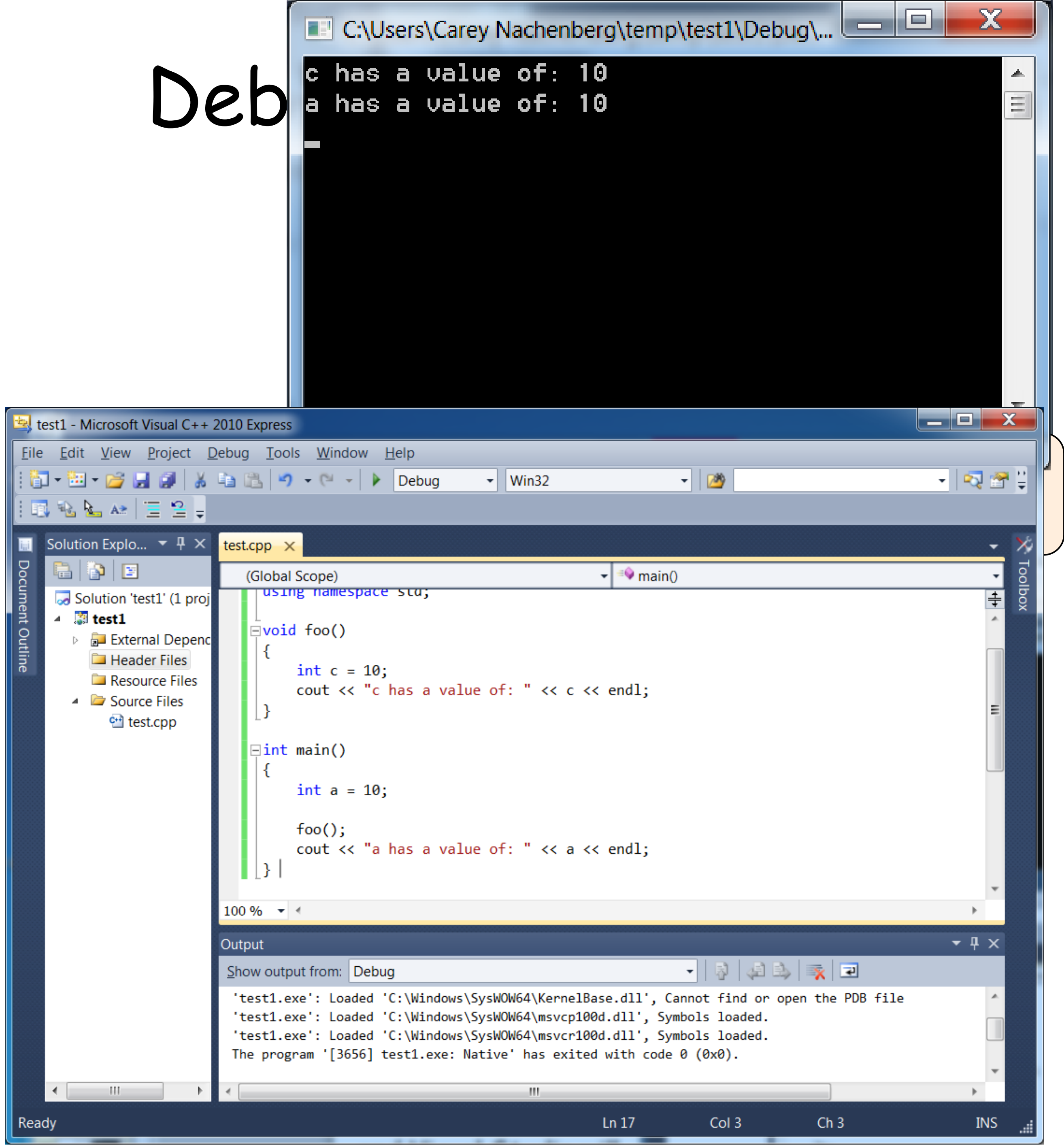
Deb



If you hit F11 while on a function call line, Visual Studio will let you step into that function and trace through it a line at a time.

So F11 lets you dive into a function call and trace through it line by line as well.

Deb



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!

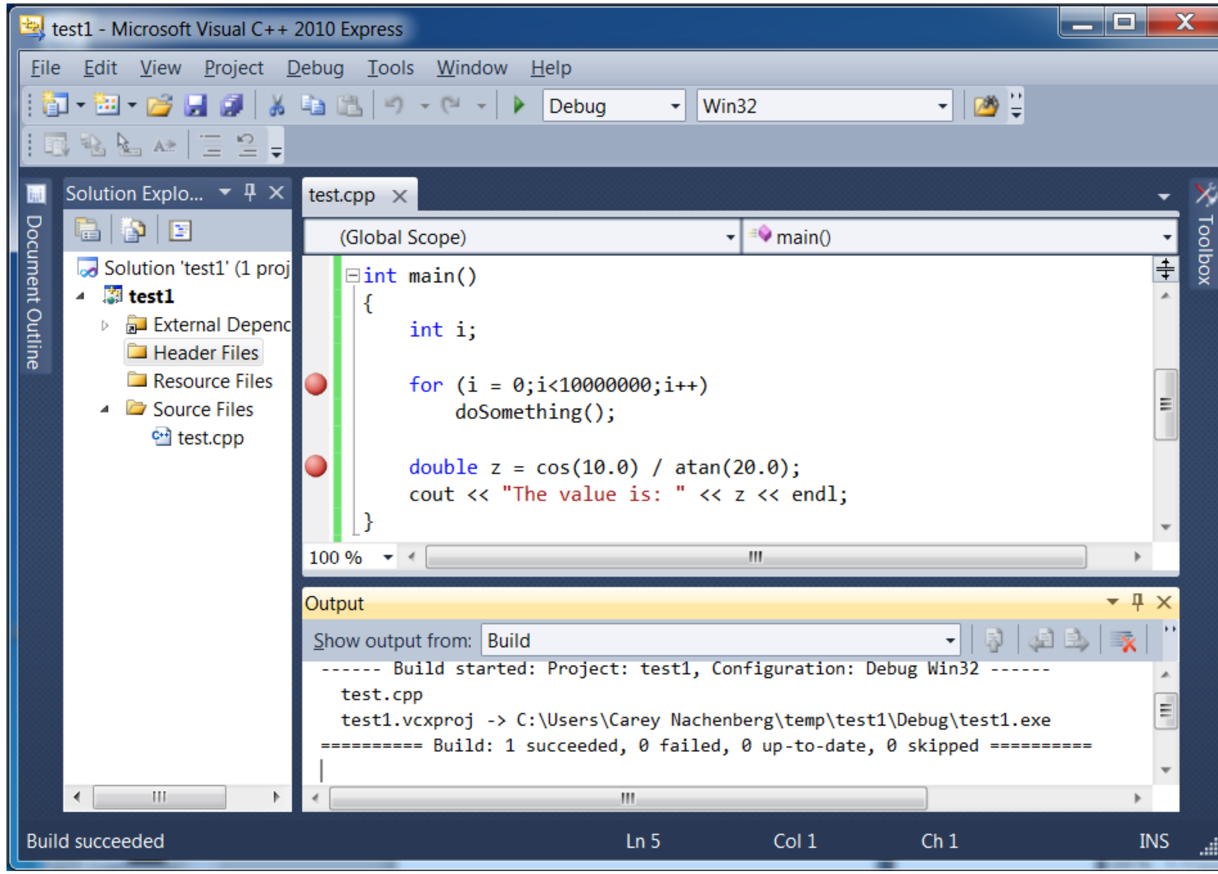
Debugging

Sometimes you'll want to start debugging deep within your program.

You don't want to trace through millions of lines to reach the likely location of a bug.

What should you do?

Answer:
Create a breakpoint!



Go to the line where you'd like to start debugging and hit the F9 key.

Then just hit the F5 key (not Ctrl-F5) to run your program until it reaches that line!

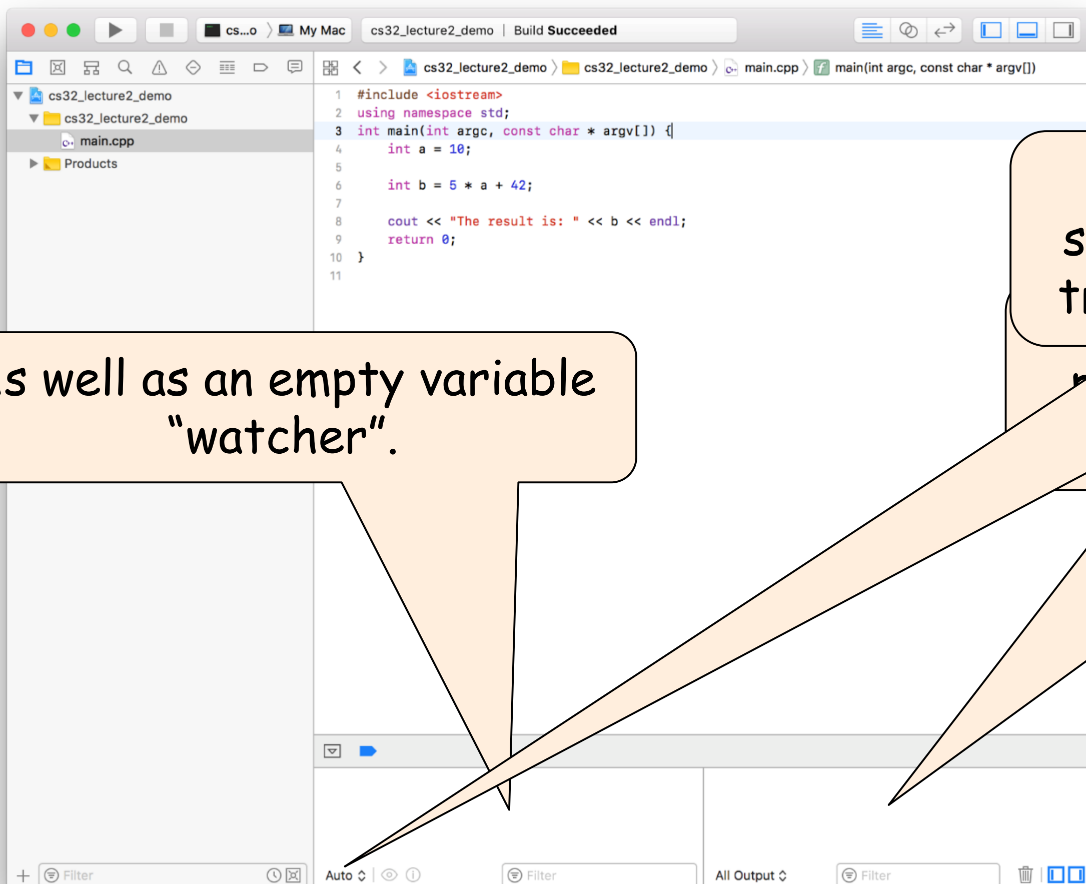
Debugging with Xcode

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.

you'll see an empty debug output window…

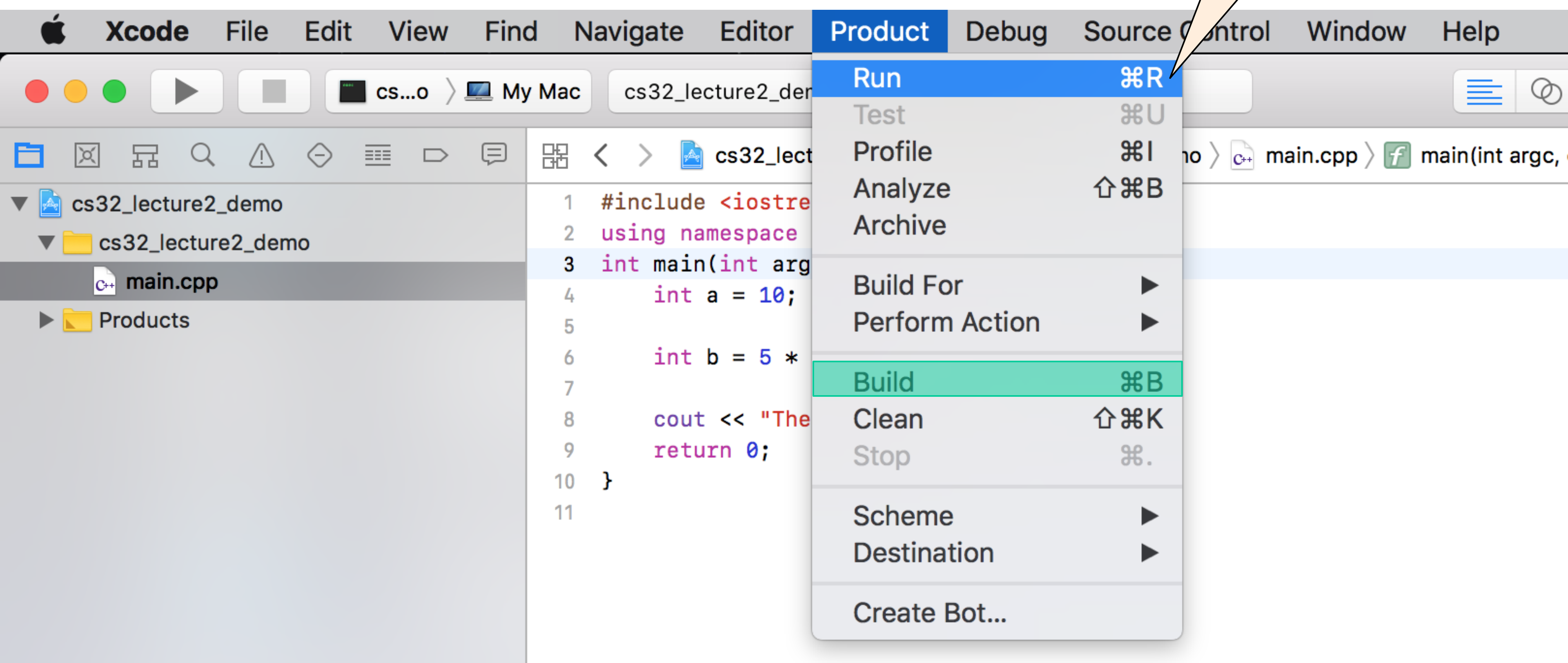
As well as an empty variable "watcher".

Debugging with Xcode

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

Product > Run

Xcode also has shortcuts ☺



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**.

```
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 }
```

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 }
11
```

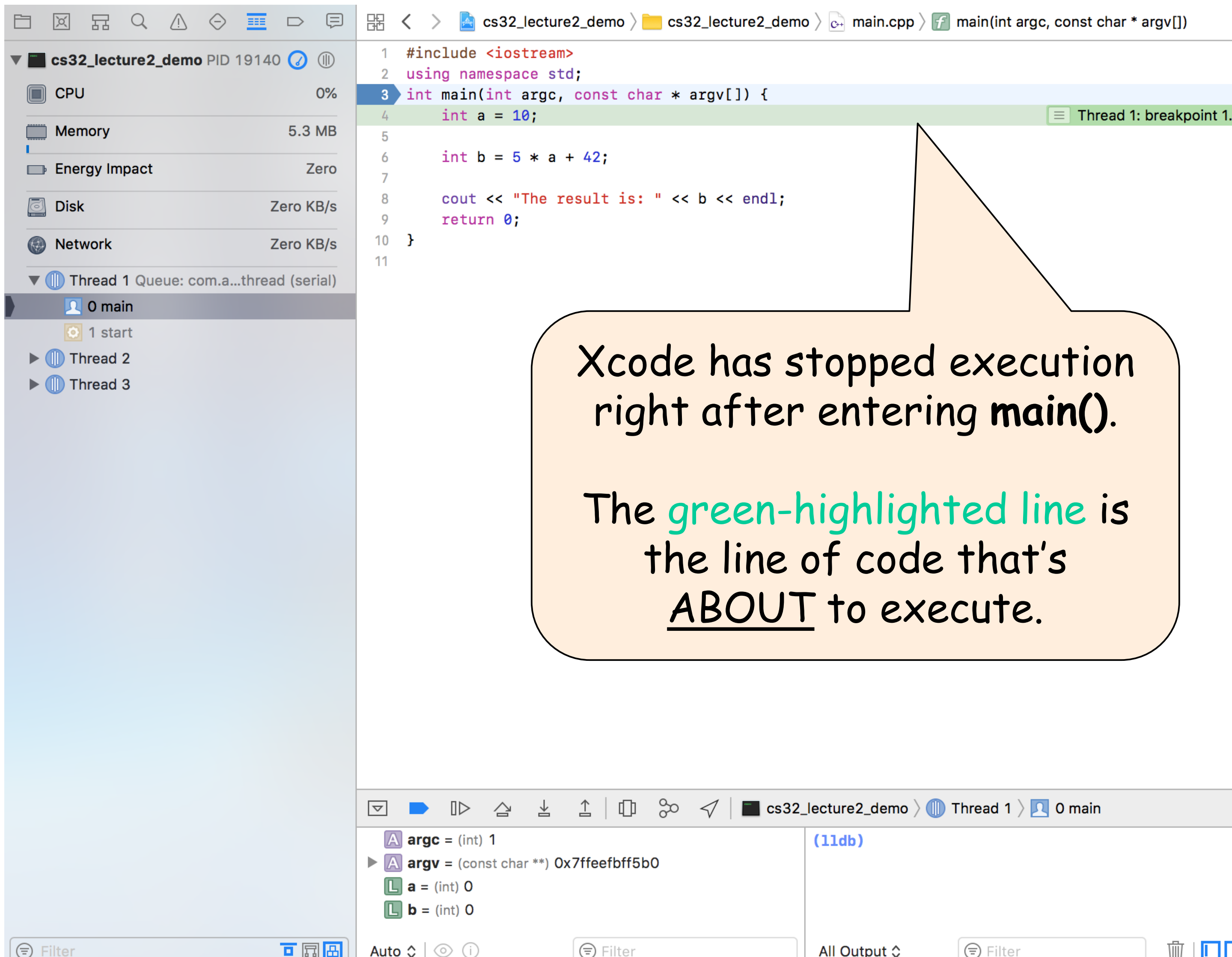
Debugging with Xcode

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.

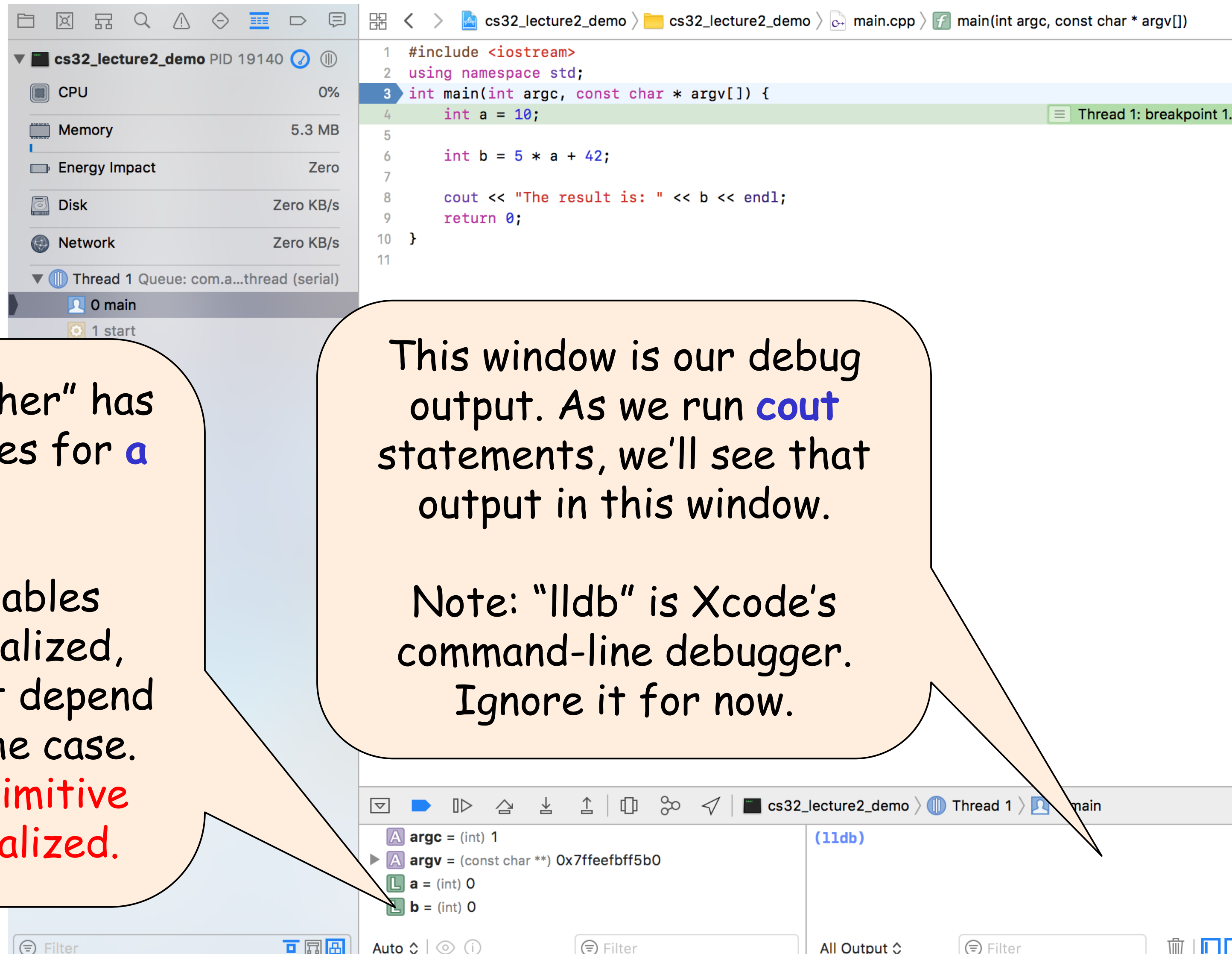


Xcode has stopped execution right after entering **main()**.

The green-highlighted line is the line of code that's ABOUT to execute.

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**.

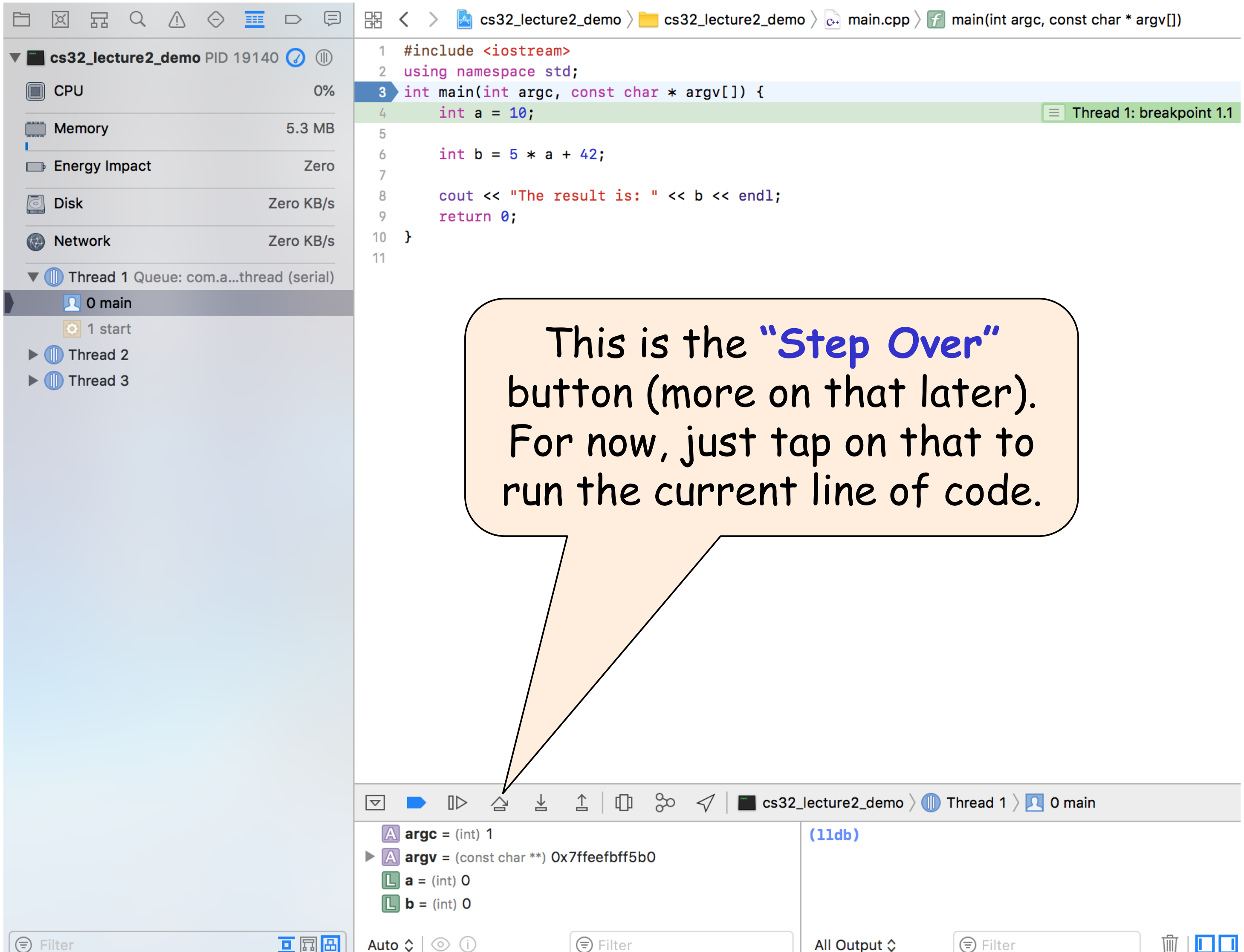
The a and b variables appear to be initialized, but in C++ you can't depend upon that to be the case. Always assume primitive values are uninitialized.

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.

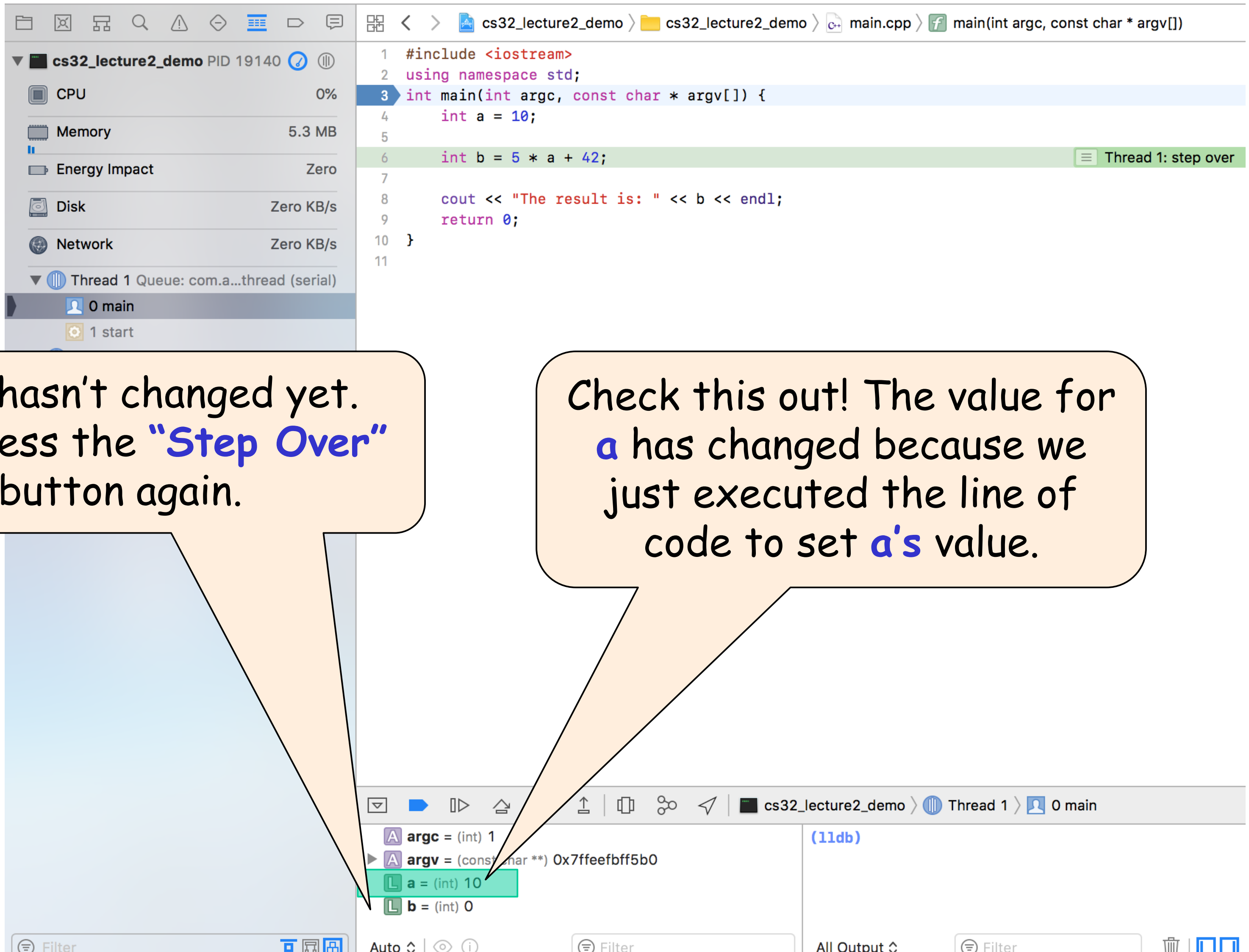
Debugging with Xcode

Let's run the current line of code.



Debugging with Xcode

Let's run the current line of code.

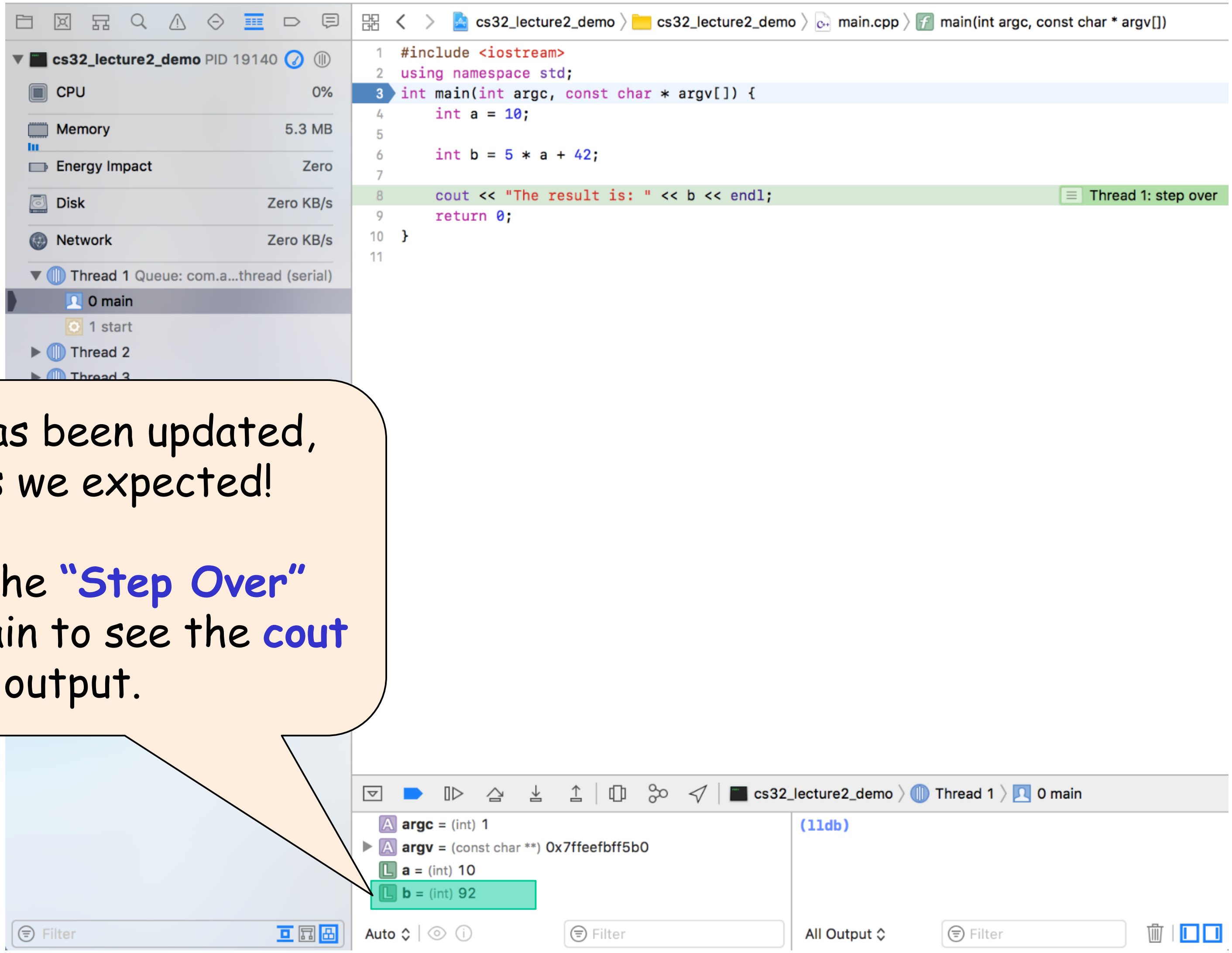


But **b** hasn't changed yet. Let's press the **"Step Over"** button again.

Check this out! The value for **a** has changed because we just executed the line of code to set **a's** value.

Debugging with Xcode

Let's run the current line of code.

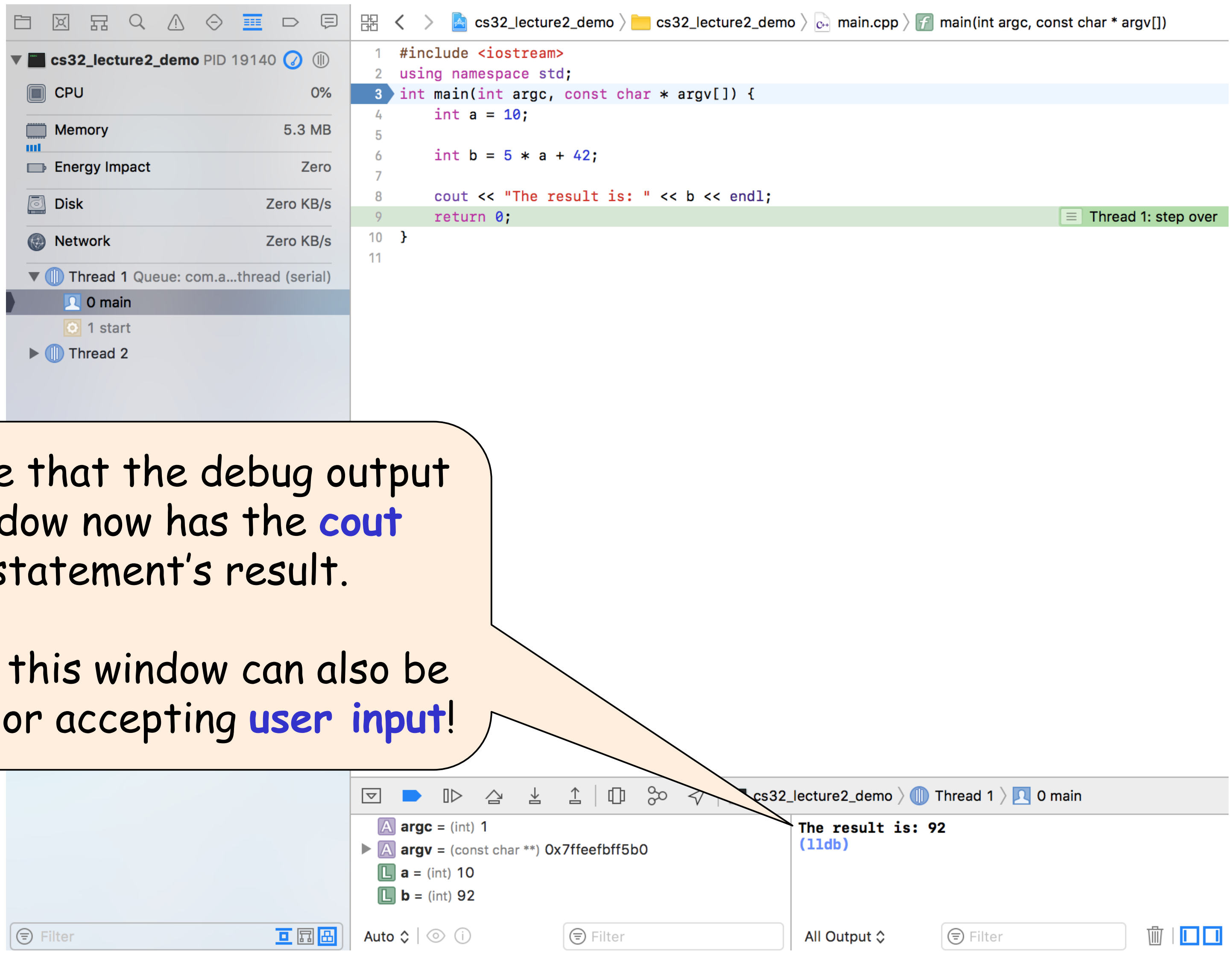


Now **b** has been updated, just as we expected!

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

Debugging with Xcode

Let's run the current line of code.



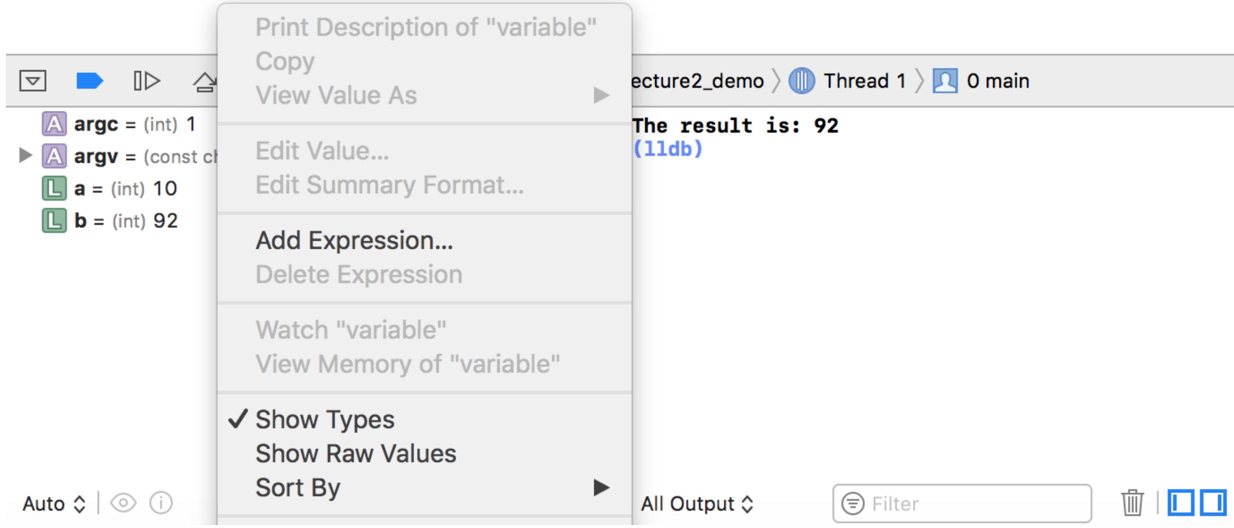
Notice that the debug output window now has the **cout** statement's result.

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

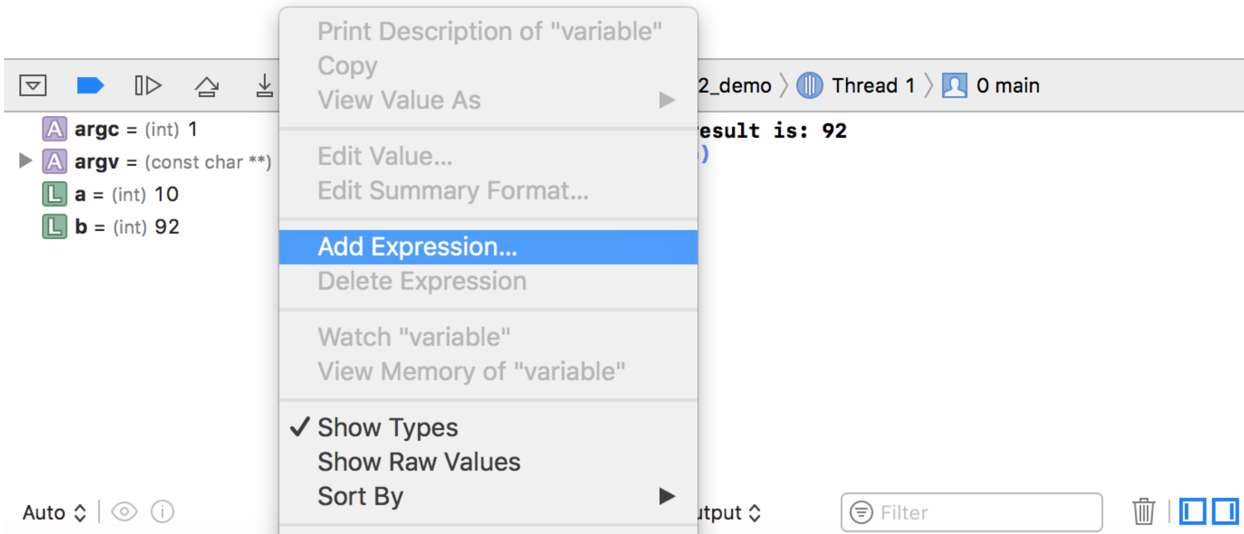
Debugging with Xcode

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

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

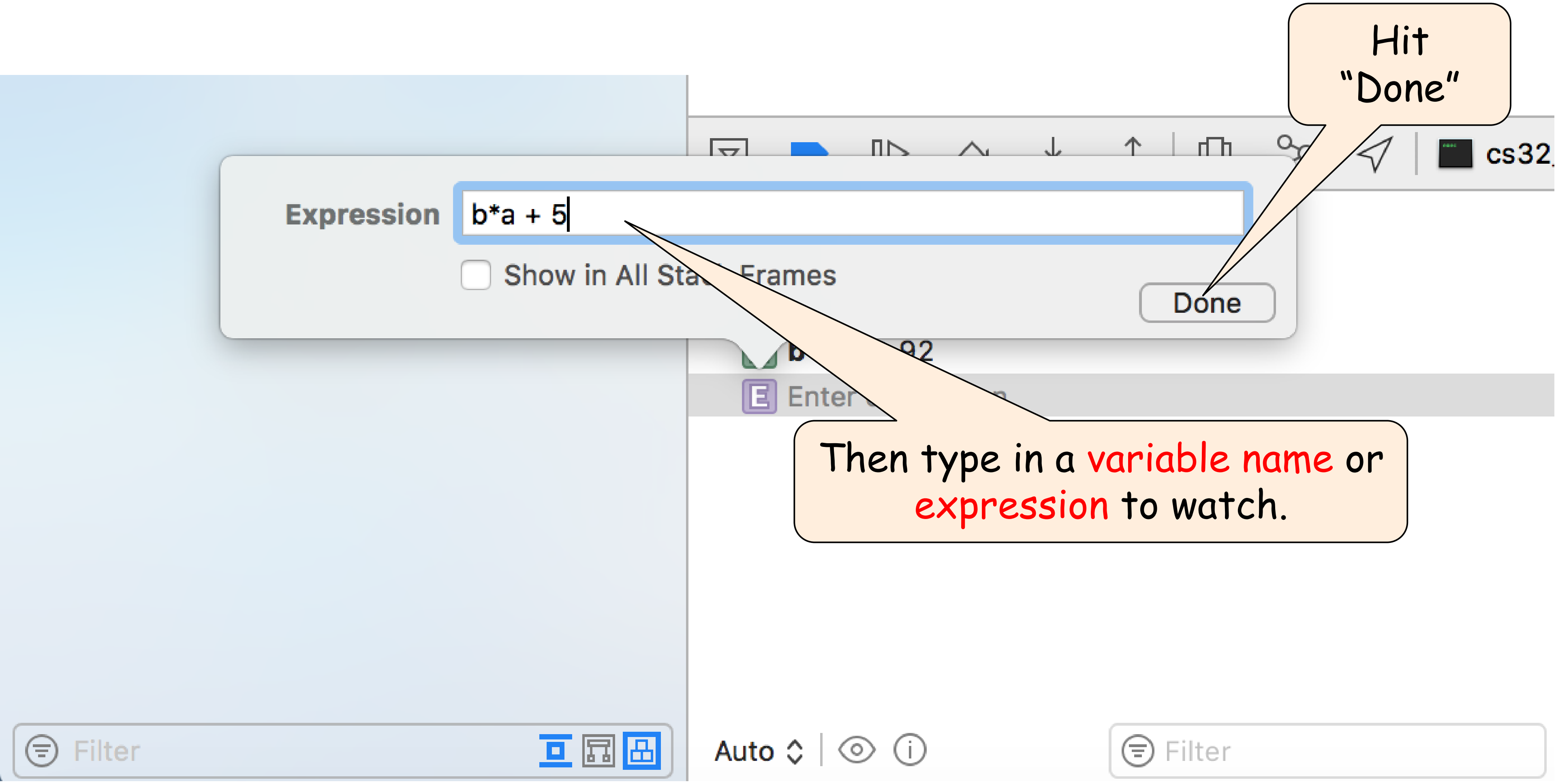


Now click on **Add Expression...**



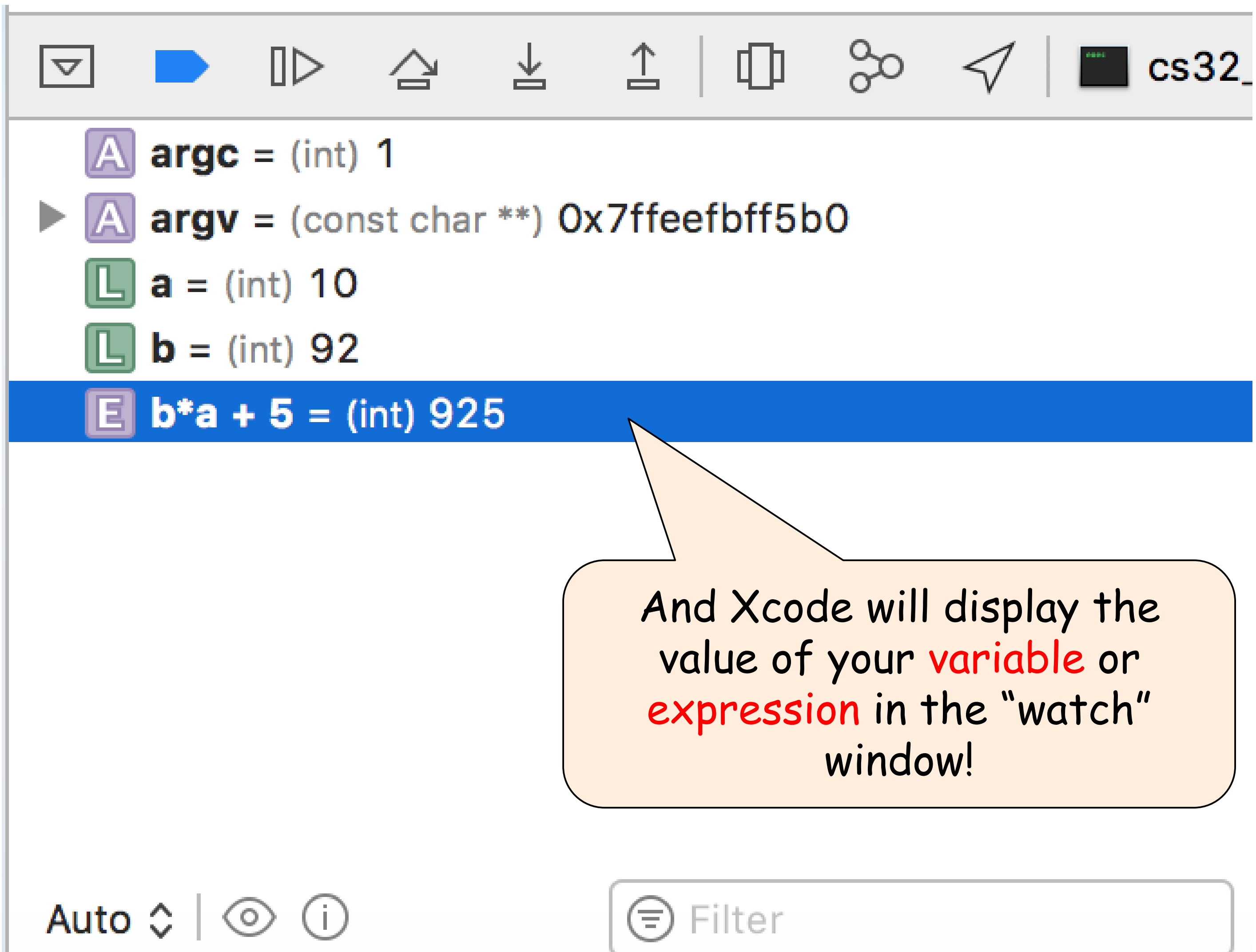
Debugging with Xcode

Add an expression to watch



Debugging with Xcode

Add an expression to watch



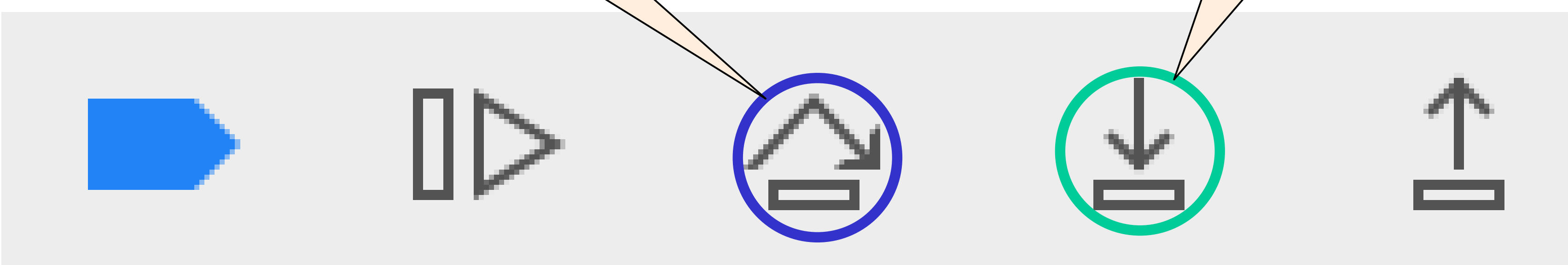
Debugging with Xcode

Let’s revisit the “Step Over” button. Note the other button right next to it.

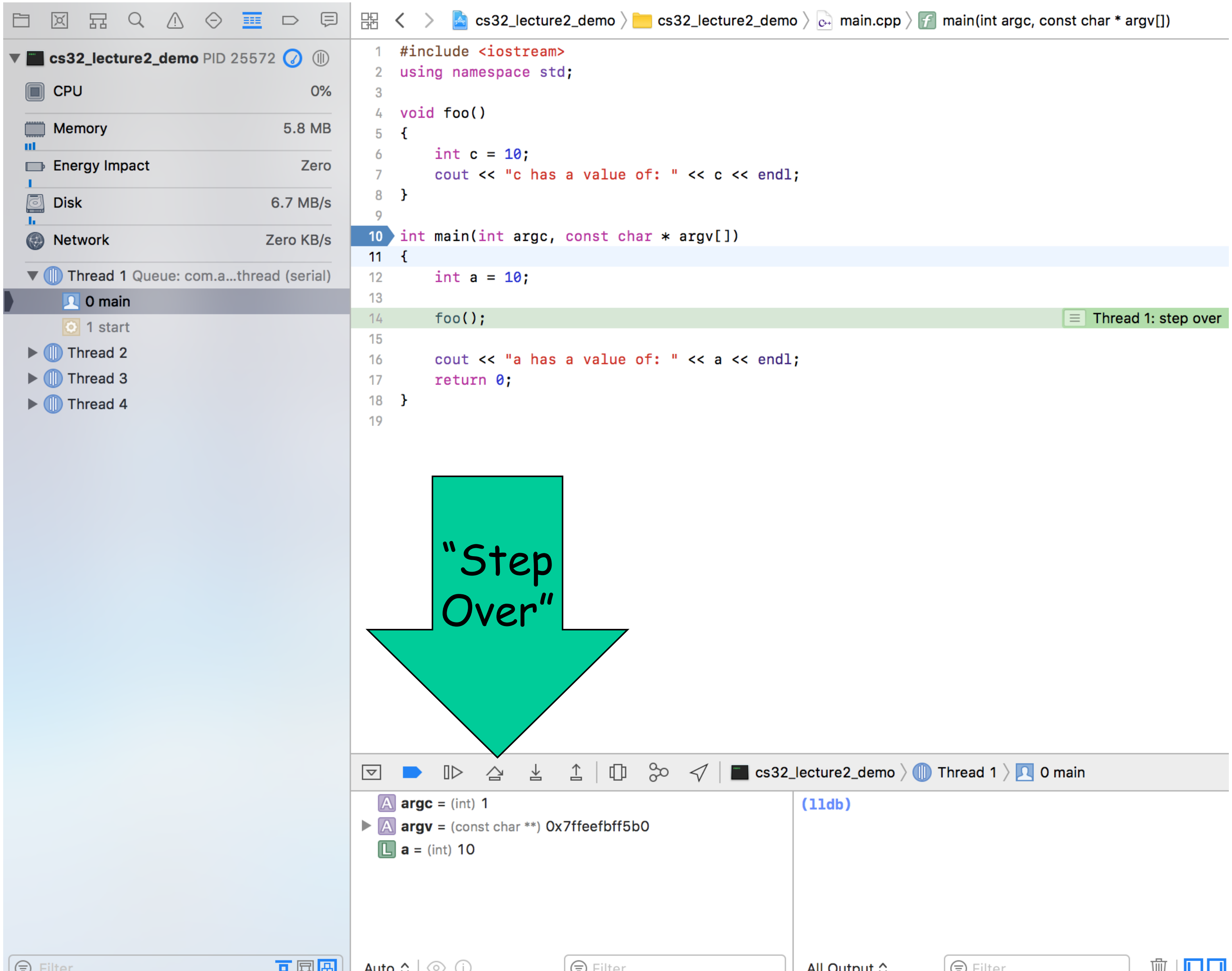
This is the “**Step Over**” button (the button we have been using so far).

Step Over” and “Step Into” tons do slightly different ngs. Let’s see an example.

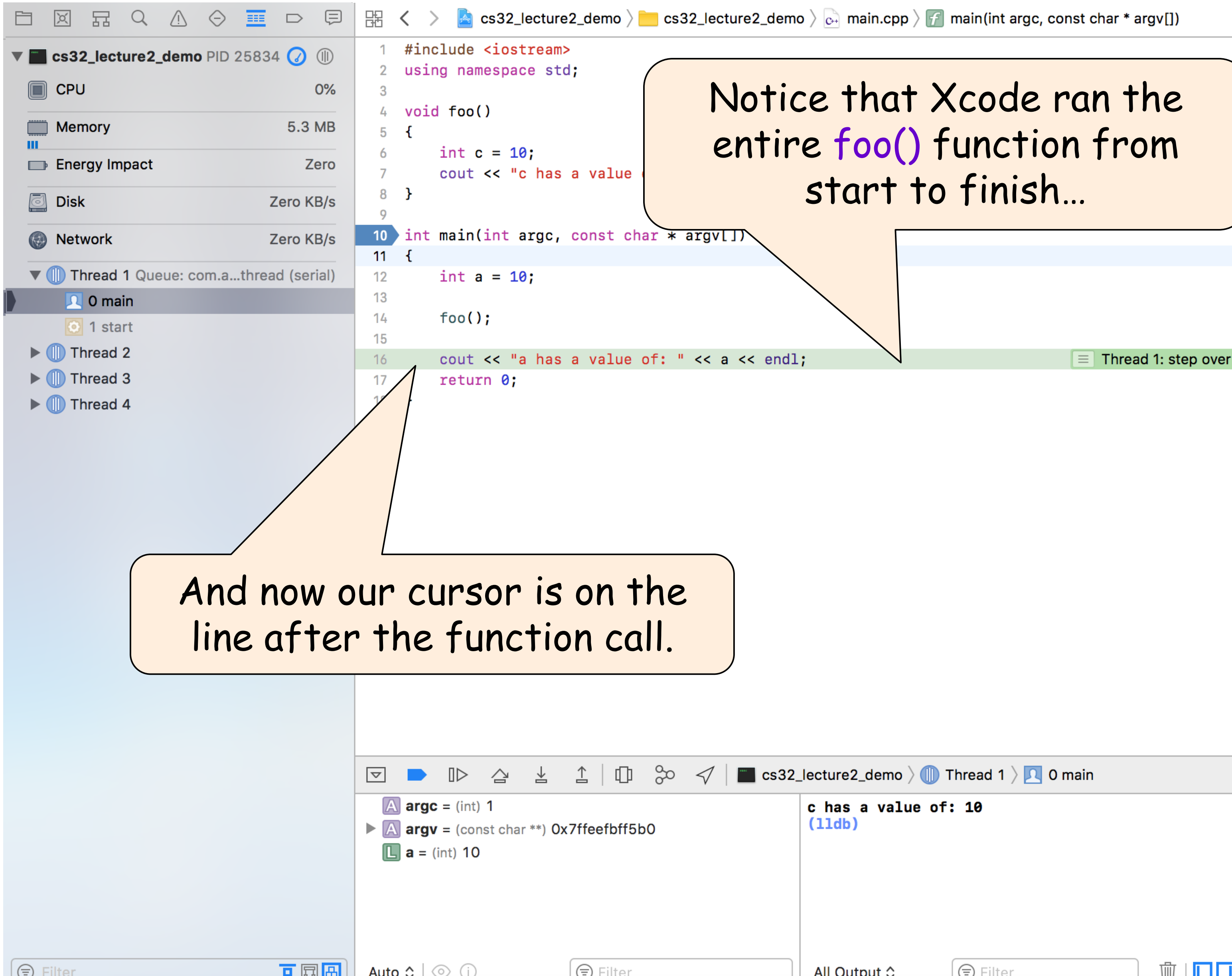
This is the “**Step Into**” button.



Debugging with Xcode



Debugging with Xcode



Notice that Xcode ran the entire foo() function from start to finish...

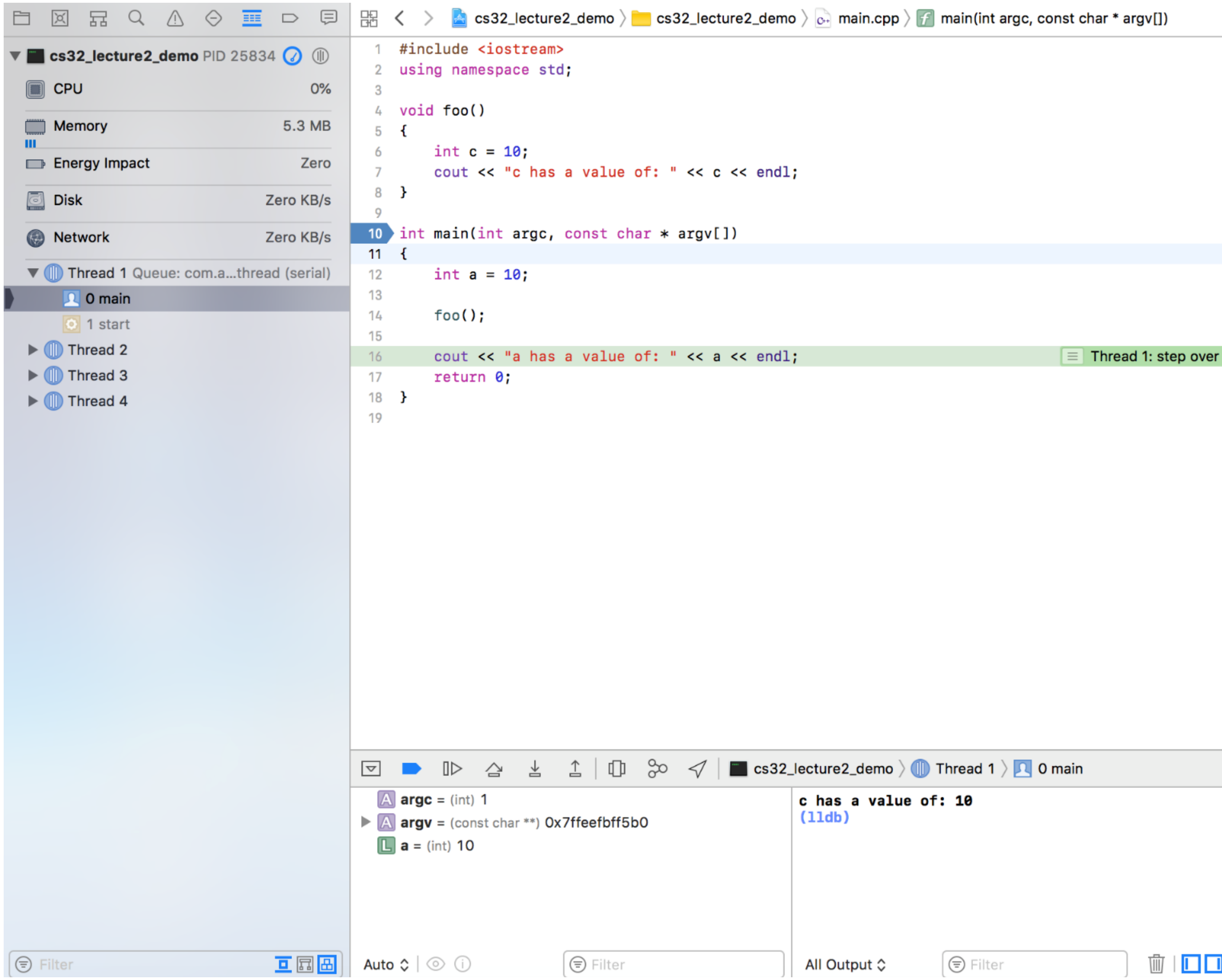
And now our cursor is on the line after the function call.

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.

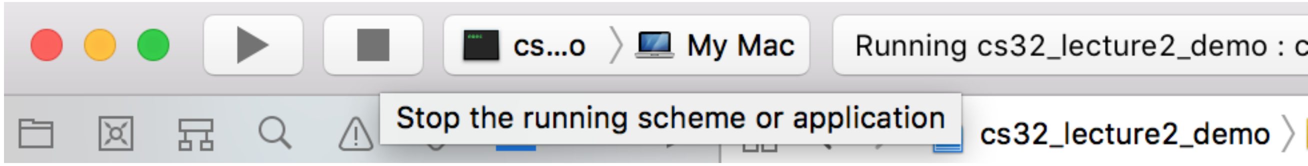
So "Step Over" lets you quickly run a line of code without delving into the details.



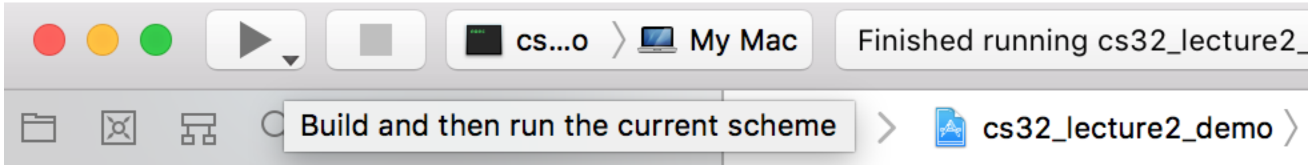
Debugging with Xcode

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

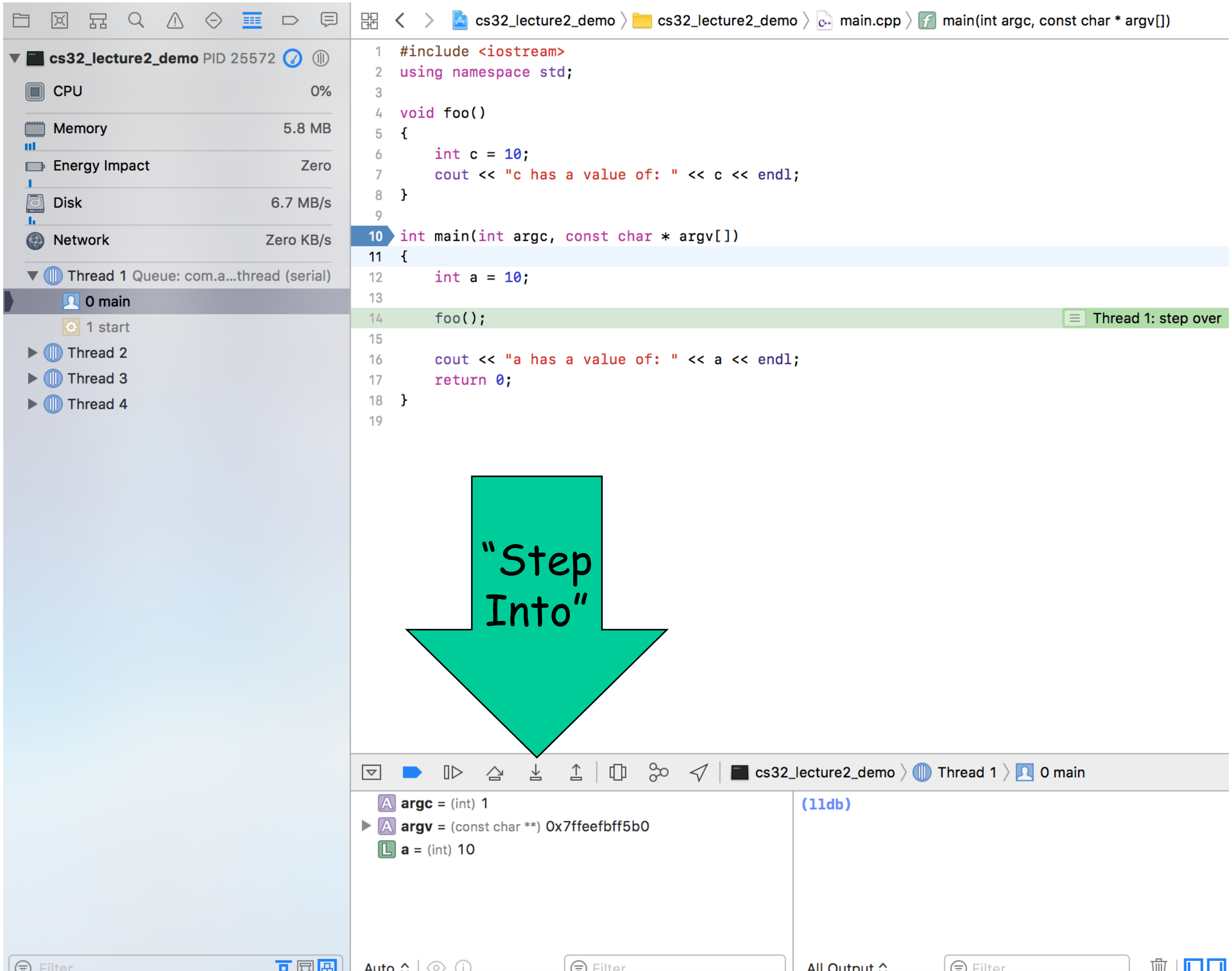
Just tap on the **Stop** button to stop the current debugging session.



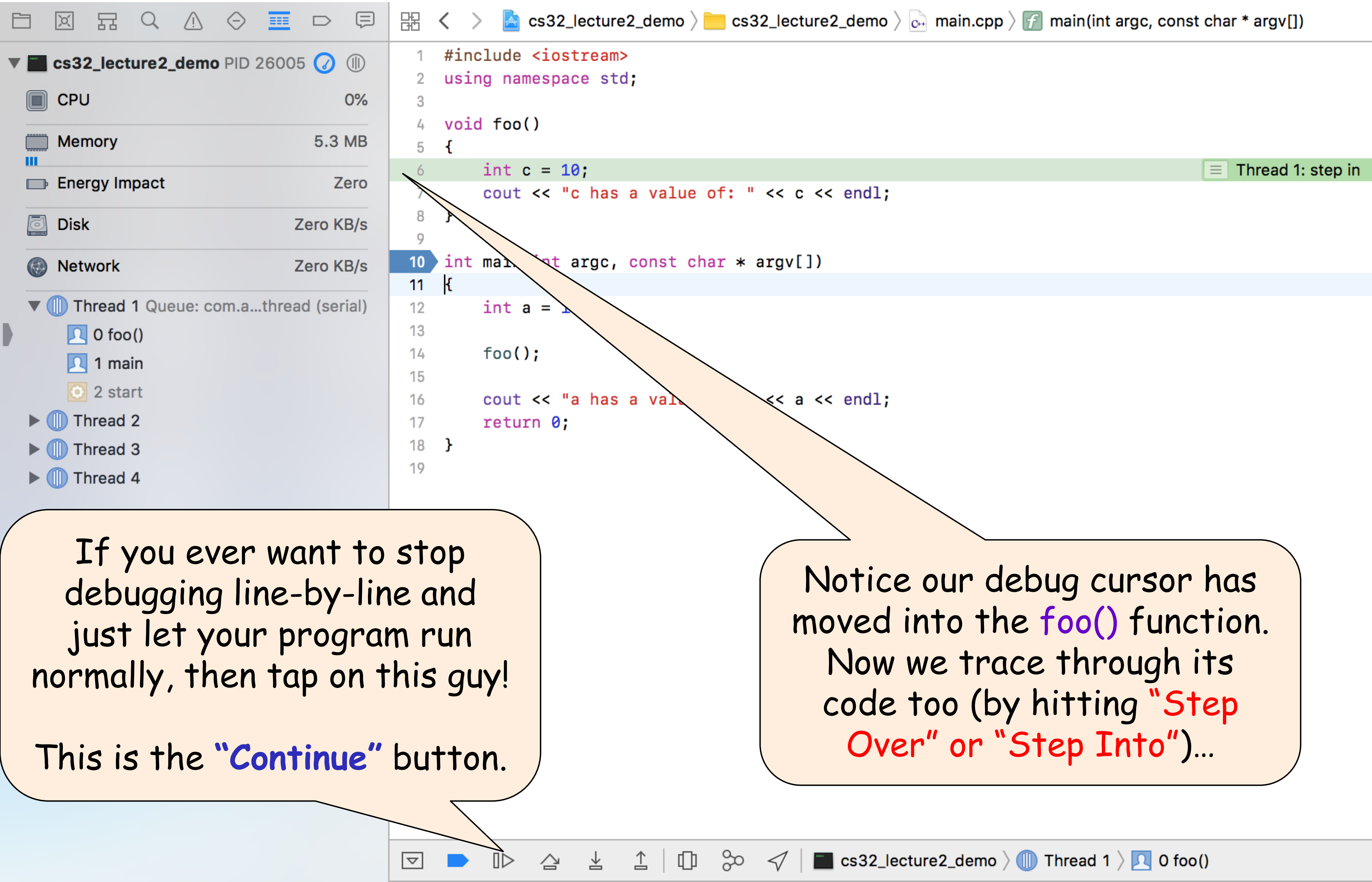
Then tap on the **Run** button to re-build and run the code.



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.

```
int main(int argc, const char * argv[])
{

    foo();

    for (int i=0; i<1000000; i++)
    {
        cout << "I'm wasting my time debugging a for loop" << endl;
    }

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

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

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!

```
int main(int argc, const char * argv[])
{

    foo();

    for (int i=0; i<1000000; i++)
    {
        cout << "I'm wasting my time debugging a for loop" << endl;
    }

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

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

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

After printing out a million of these lines ☺

```
int main(int argc, const char * argv[])
{

    foo();

    for (int i=0; i<1000000; i++)
    {
        cout << "I'm wasting my time debugging a for loop" << endl;
    }

    double z = cos(10.0) / atan(20.0);          Thread 1: breakpoint 1.1

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

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

file2.cpp

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

void otherFunc(void)
{
   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

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

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;
}
```

What can you do?
Always DIRECTLY include the header files you need RIGHT where you need them. In this case, main.cpp has a Student variable and an Alcohol variable. So it should include both of their header files.

main.cpp defines an Alcohol variable but it doesn't #include"alcohol.h".

Why? It assumes that student.h will just include alcohol.h for it.

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:

file.cpp

```
#define PI 3.14159

#define FOOBAR

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

You can also use #define to define a new constant without specifying a value! Why you ask? We'll see!

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.

file.cpp

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

Preprocessor Directives

Command 2: #ifndef and #endif

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.

file.cpp

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

Separate Compilation

When using class composition, it helps to define each class in a separate pair of .cpp/.h files.

Then you can use them like this in your main program…

Now – something interesting (and bad) happens if I use both classes in my main program. Let's see!

Can anyone see what the problem is?

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();
}
```

Separate Compilation

Your main program first includes student.h...

Then student.h then includes calc.h!

...and finally, main.cpp includes calc.h... again!

So what's the problem?
Well, since main.cpp and student.h both included calc.h, we ended up with two definitions of Calc! That's bad!

This can result in a compiler error!

So how do we fix it, you ask?

Here's how...

main.cpp

```
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();
}
```

Separate Compilation

calc.h

```
#ifndef CALC_H

#define CALC_H


class Calculator
{
public:
   void compute();
   ...
};

#endif // for CALC_H
```

Add "include guards" to each header file.

An include guard is a special check that prevents duplicate header inclusions.

Now, when the compiler compiles this code, it will ignore the redefined definitions!

student.h

```
#ifndef STUDENT_H
#define STUDENT_H

#include "calc.h"

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

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

alcohol.h

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

student.h

```
#include "alcohol.h"

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

So I need to include alcohol.h, right?

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

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

Well, not so fast!

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

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];
};
```

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!

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).

On the other hand…

A.h

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

B.h

```
class FirstClass;

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];
};
```

This line tells C++ that your class exists, but doesn't actually define all the gory details.

Since none of this code actually uses the "guts" of the class (as the code did on the previous slide), this is all C++ needs to know!

Last Topic: Knowing HEN to Include .H Files

If you do NONE of the previous items, but you...

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 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).

Here's how you do that:

A.h

```
class FirstClass
{
public:
  // thousands of lines
  …void someFunc() {…}
  // 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?

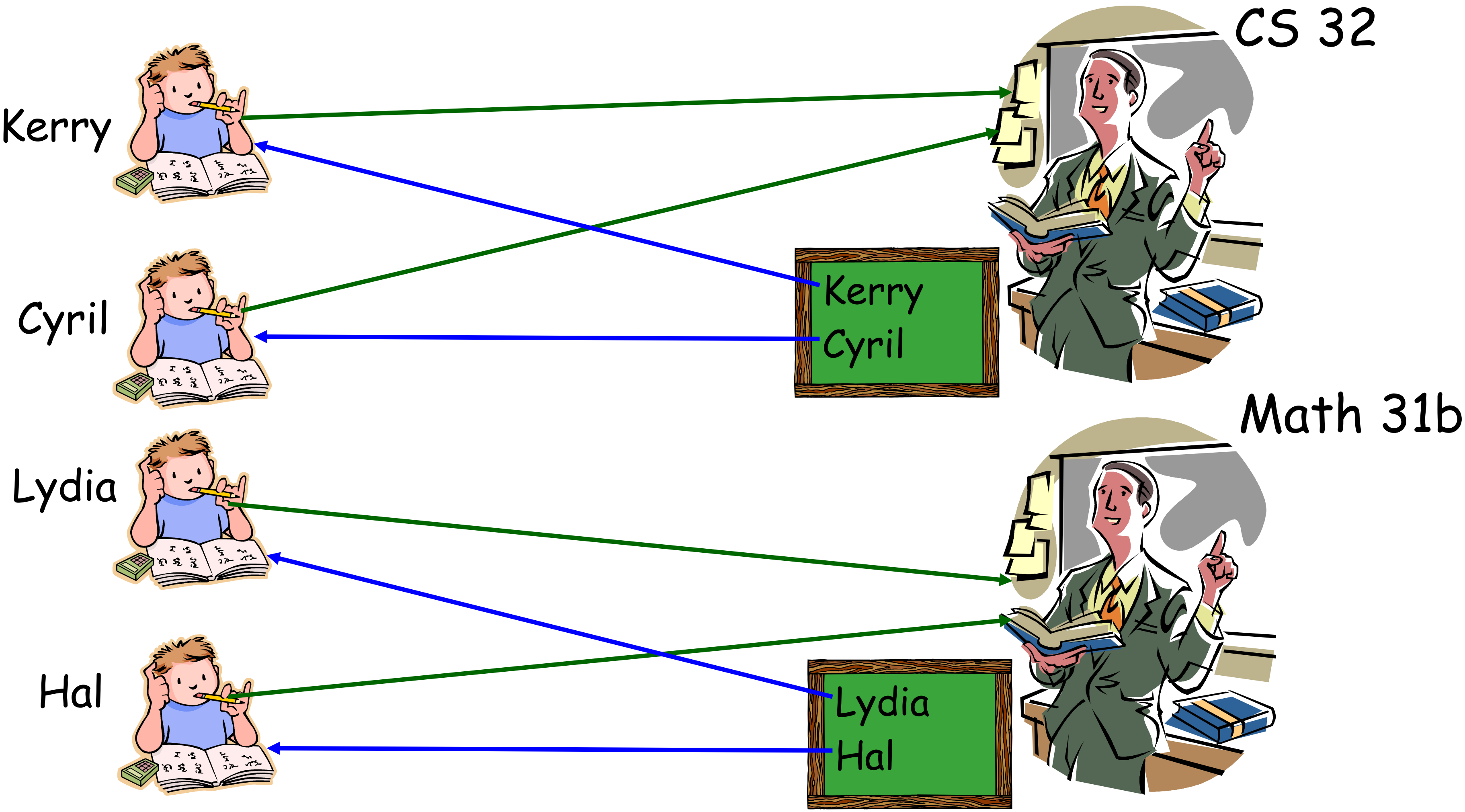
There are two reasons:

1. If a .h file is large (thousands of lines), #including it when you don't strictly need to slows down your compilation!
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.

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

Here we have a ClassRoom class that holds a bunch of Students…

And here we have the Student class. Each Student knows which classroom he or she is in…

Since our ClassRoom class refers to the Student class it includes Student.h

Since our Student class refers to the classroom class, it includes ClassRoom.h…

Do you see the problem?

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();
}
```

Last Topic: Self-referential Classes

main.cpp

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

Student.h

```
#include "ClassRoom.h"

                        m;
```

The compiler will keep going forever!!!!!

(oh, and adding Include Guards doesn't solve the underlying problem either!)

```
public:
  ...

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

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?

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!!!

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

Student.h

```
#include "ClassRoo

class Student
{
public:
    ...

private:
    ClassRoom *m_myRoom;
};
```

ClassRoom.h

```
#include "Stude    "

class ClassRoom
{
public:
    ...

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

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).

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 class'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 one or more functions DIRECTLY in your class definition AND they use your other class in a significant way, then you must MOVE them to the CPP file.

Student.h

```
class ClassRoom;

class Student
{
public:
 void beObnoxious();

private:
  ClassRoom *m_myRoom;
};
```

Student.cpp

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

 void Student::beObnoxious() {
   cout << m_myRoom->getRmNum() << " sucks!"; }


}
```

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"   );
    poop("Sylvia"   );
    poop("Carey",3);
    poop("David",0);
    poop("Sergey"   );
    poop("Larry"   );
    poop("Devan"   );
}
```

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)
```

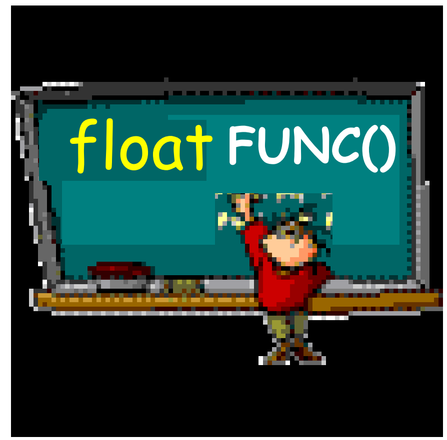
```
// PERFECT! All default parameters come after non-defaults
bool burp(int length, int loudness = 5, int pitch = 60)
{
   ...
}
```

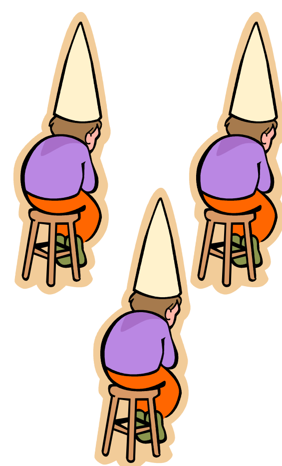
Class Challenge

RULES

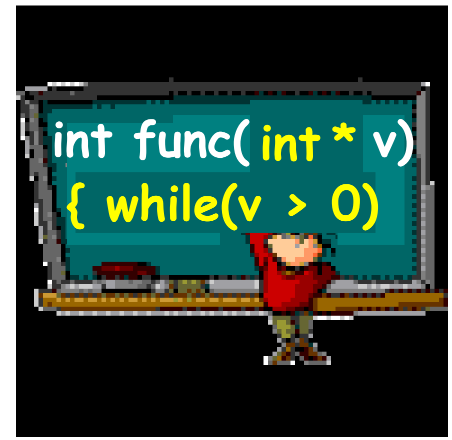
- The class will split into left and right teams
- One student from each team will come up to the board
- Each student can either
  - write one new line of code to solve the problem OR
  - fix a single error in the code their teammates have already written
- Then the next two people come up, etc.
- The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

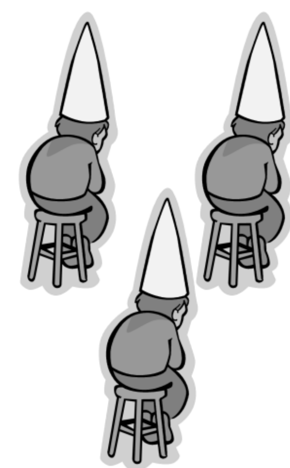
Team #1





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 class, 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.