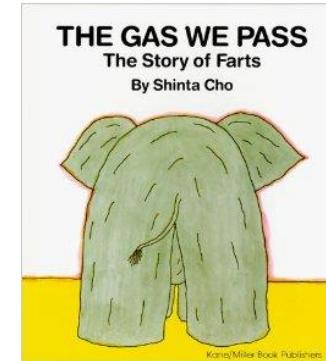


Lecture #2

(aka the Fart Lecture)

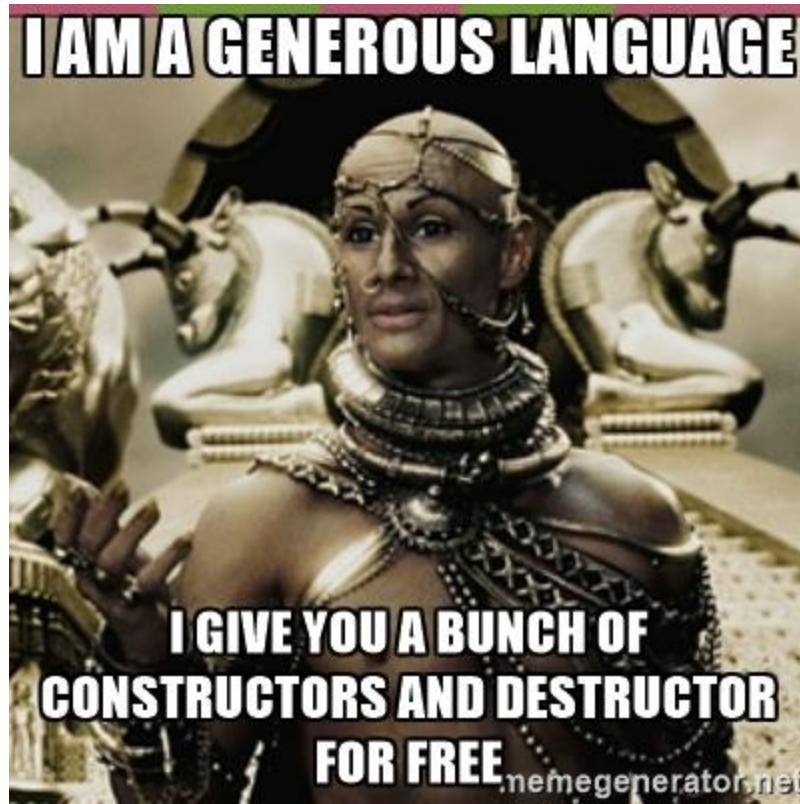
- Part 1: Basic C++ Concepts
 - Constructors
 - Destructors
 - Class Composition
 - Composition with Initializer Lists
- Part 2: Becoming a Debugging Ninja
 - Learning how to use the Visual C++ debugger
- Part 3: On-your-own Study Topics
 - A few final topics you need to know to do Project #1



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!

Constructors, Destructors, etc.



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 Init

Every class should have an **initialization function** that can be used to reset new variables before they're used.

2 false

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

What is it?

```
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

```
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);
    }
}
```

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

Ugh! Remember, all simple 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...**

```
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;
};
```

Constructors

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();
}
```

```

class Gassy
{
public:
    Gassy (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;
};

```

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

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();
}

```

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

```

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

```

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

The constructor has no return type...
Not **void**, **int** or **bool**!

ctors

```
class Gassy
{
public:
    Gassy(int age, bool ateBeans)
    {
        m_age = age;
        m_ateBeans = ateBeans;
        return(true); //illegal!
    }
    ...
private:
    int m_age;
    bool m_ateBeans;
};
```

The constructor always has the same name as the class it's in.

Thus, it's not allowed to return any **value** at all!

You can define the constructor in the **class declaration** (see above)...

```
class Gassy
{
public:
    Gassy(int age, bool ateBeans);
```

And remember to also include just the **function header**, followed by a **semicolon**, in the class declaration itself.

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

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

Boy, isn't that syntax ugly? Just remember, to define a constructor outside the class declaration:

use the **class name**, followed by **::**, followed by the **class name**

Or, **outside the class declaration**, like this...

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 lynn(18, false);
    Gassy dave; // OK!!!
}
```

Constructors

```
class Gassy
{
public:
    Gassy() // generated by compiler
    {
        // I don't initialize scalar member
        // variables... So be careful!!!
    }

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

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



These variables will have random values!

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

Non-scalars include `string`, `Gassy`, etc.

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 **scalar** member variables!

So be careful!

```
int main()
{
    Gassy carey;
    cout <<
        carey.getFartsPerHr(); //??
}
```

Constructors & Arrays

```
class Gassy
{
public:
    Gassy(int age, bool ateBeans)
    {
        m_age = age;
        m_ateBeans = ateBeans;
    }
    Gassy()
    {
        m_age = 1;
        m_ateBeans = false;
    }
    ...
private:
    int m_age;
    bool m_ateBeans;
};
```

The constructor is then run on *every* element in the array!

Your class **must** at a minimum have a constructor that requires no arguments!

lec1
[0]
[1]
[2]
[3]

m_age	1
m_ateBeans	false
m_age	1
m_ateBeans	false
m_age	1
m_ateBeans	false
m_age	1
m_ateBeans	false

If you want to have an **array** of your class...

```
int main()
{
    Gassy lec1[4];
    ...
}
```

When Constructors

```
int main()
{
```

```
Gassy carey(45, false), bill;
```

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

```
Gassy arr[52];
```

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

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

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

```
Gassy *justAPtr;
```

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

```
}
```

Class Challenge

Name all the times the Gassy constructor is called in this example...

```
class Gassy
{
public:
    Gassy()
    {
        m_age = 0;
        m_ateBeans = true;
    }
    ...
private:
    int m_age;
    bool m_ateBeans;
};
```

```
void foo()
{
    Gassy *herbert = new Gassy;
}

int main(void)
{
    int j;
    Gassy *ptrToNerd;
    Gassy xavier;

    for (j=0;j<3;j++)
    {
        Gassy a[5];
        foo();
    }
}
```

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;
SDL_Surface*U, * V;
){ u=C+X *7; b*u){                                     ;
}; H=u          [2]; z           =(H&8)          ;
i; A=(          H/B)*i          ; B=u[          ;
]/**/;          return          u; }          ;
0; } Y          ()( O=50; r=          0; t=1
) Z (X          ,m, n, o)/**/          { /**
return          W(X)&&B&&(m<
y+1&x<m+1&x
,bb=C; a(X,
=bb +1; H=6;
N
E
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=Ba1; b D&bg) continue;
m-=be; n-=bf; b Ba8){ u[-i]=0; j=1; bc(8); b Ba32){ n-= i;
o=i*2; } b Ba16&&O){ bc(32+(o>i?8:0)); Y(); u[- 1]=0; }
b(Ba128&&bb&&s>0)||Ba64))u[-1]=0; 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+=i-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;
} } b(m,e,k,         bi){ H=x/ 2;         G[bi]=m>e-H?
e:(m>H?H-m: 0 ); } } b j(X,be         ,bf){ int bk
u[0]+=be; u[1]         +=bf; W(X);         E=x,F=y,I=0;
(         bk,0){ b I=(Xl=bk&&Z(bk,E,F,i)&&(Ba6
)break; } W(X); b I){ b bf)u[1]=-bf;
be         ){ u[0]=-be; u[4]*=-1*be; } } W( X);
bl         (){ int bm=n,X; SDL_FillRect(U,0,M); X=4; while(
--         )bd(x,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 Ba1){ *u +=u[4]; b
++u[5]>20){ u[4]*=
#271:0; } J=i; b q
r)z+=i*(K%2); b lt)
; }} b q!=u||!(D&&
bn={ z,A,i,J} ,bo=
J}; SDL_BlitSurface
} b(s+2)>2>s; K
b!-- O) exit(L); }
(T[X],0,0); } bq(int H,
bt=Q-P; b bt>bs) bt= bs;
128); P+=bt; b P>Q)b=0;
,8,1,0,256,0,0,bq},bv ;
-bx; int by,H=255, bz=H<<8
bC=bp(5),bD=bp(6); SDL_Event
,l=bp(2),e=bp(3),h =bp (4); M=
0xffff
; b1*(char*) &N{ H=>B; bz=>A; ba
bB=255
; } U=SDL_SetVideoMode(k,l,0,0); V/*NES HISTORY
SANDRO
,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*/
"sd ", u+H++);
u)+blu [3])( q
} } for ( ; ; ){
SDL_PollEvent(&bE) ) { by
type==2){ I = bE.
key/**/
by?0:(p-1); b I== 275 )r=
0:(t?(s=-9):0); b I==27exit(0); } } bl(); SDL_Flip(U); SDL_Delay(60); } }
,ba =bz <<8,bB= ba<<8,
bE; o =i= bc / 8; k = bp(1)
bg(9); /**/ SDL_Init(/*<2053*/
&N{ H=>B; bz=>A; ba
-bz>>8;
&N{ H=>B; bz=>A; ba
-bz>>8;
*/ SDL_CreateRGBSurface(1 <<15 , bc, bd,
32,H,bz
pixels,bC*bD*4,1,fopen(T [7],"r" )); /**/ SDL_OpenAudio(&
,T[8],& bv,&S,&Q ); /**/ SDL_PauseAudio(0);
while ( H < 6) scanf(*"/0*/
"b u[5]< 0)break
=I+1; m =u[0 ];
while/* !MAFFIO
bE.type ==3; b!O &&(by|| bE.
.keysym .sym; b I== 276)r=
by?0:(p -1); b I==32)by?
```

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 **destruct** 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:
    ...
};
```

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

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**.

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

You can also define a **destructor** outside your class definition...

```
class SomeClass
{
public:
    ~SomeClass ()
    {
        // your destructor
        // code goes here
    }

private:
    ...
};
```

Destructors must
NOT have any
parameters.
Destructors must
NOT return a
value either.

Then define your actual
destructor function outside
the class like this...

```
class SomeClass
{
public:
    ~SomeClass ();
    ...

};

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

First, define a function header for
your **destructor** inside your class.
Don't forget the **semicolon!**

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?

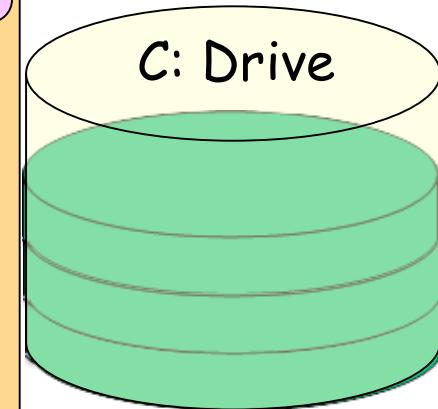
Notice - our class doesn't have a destructor. Let's see what happens!

```
class MP3Player
{
public:
    MP3Player()
    {
        reserveSpaceOnDisk(100000000); // 100MB
    }

    void getSong(const std::string &songURL)
    {
        downloadToReservedSpace(songURL);
    }

    void playSong()
    {
        playMP3InReservedSpace();
    }

    void doneWithMusic()
    {
        freeReservedSpaceOnDisk();
    }
};
```



```
int main()
{
    string url = "www.music.com/song.mp3";
    // obsessively play song 100 times!
    for (int i=0;i<100;i++)
    {
        MP3Player p;
        p.getSong(url);
        p.playSong();
    }
}
```

Why Do We Need Destructors?

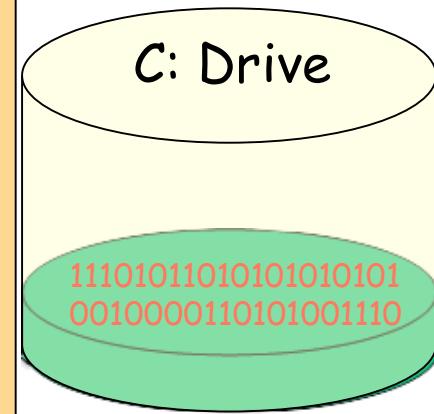
```
class MP3Player
{
public:
    MP3Player()
    {
        reserveSpaceOnDisk(100000000); // 100MB
    }

    void getSong(const std::string &songURL)
    {
        downloadToReservedSpace(songURL);
    }

    void playSong()
    {
        playMP3InReservedSpace();
    }

    void doneWithMusic()
    {
        freeReservedSpaceOnDisk();
    }

~MP3Player()
{
    freeReservedSpaceOnDisk();
}
};
```



```
int main()
{
    string url = "www.music.com/song.mp3";

    // obsessively play song 100 times!
    for (int i=0;i<100;i++)
    {
        MP3Player p;
        p.getSong(url);
        p.playSong();
    }
}
```

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!**



Dest

So when is a destructor called anyway?

```
void Dingleberry(void)  
{
```

 SomeClass a;

```
    for (int j=0;j<10;j++)  
    {
```

 SomeClass b;

 // do something

}



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.

```
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

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?

```
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;
```

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!

Class Challenge #2

Name all the times the Gassy destructor is called in the code below.

```
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;
};
```



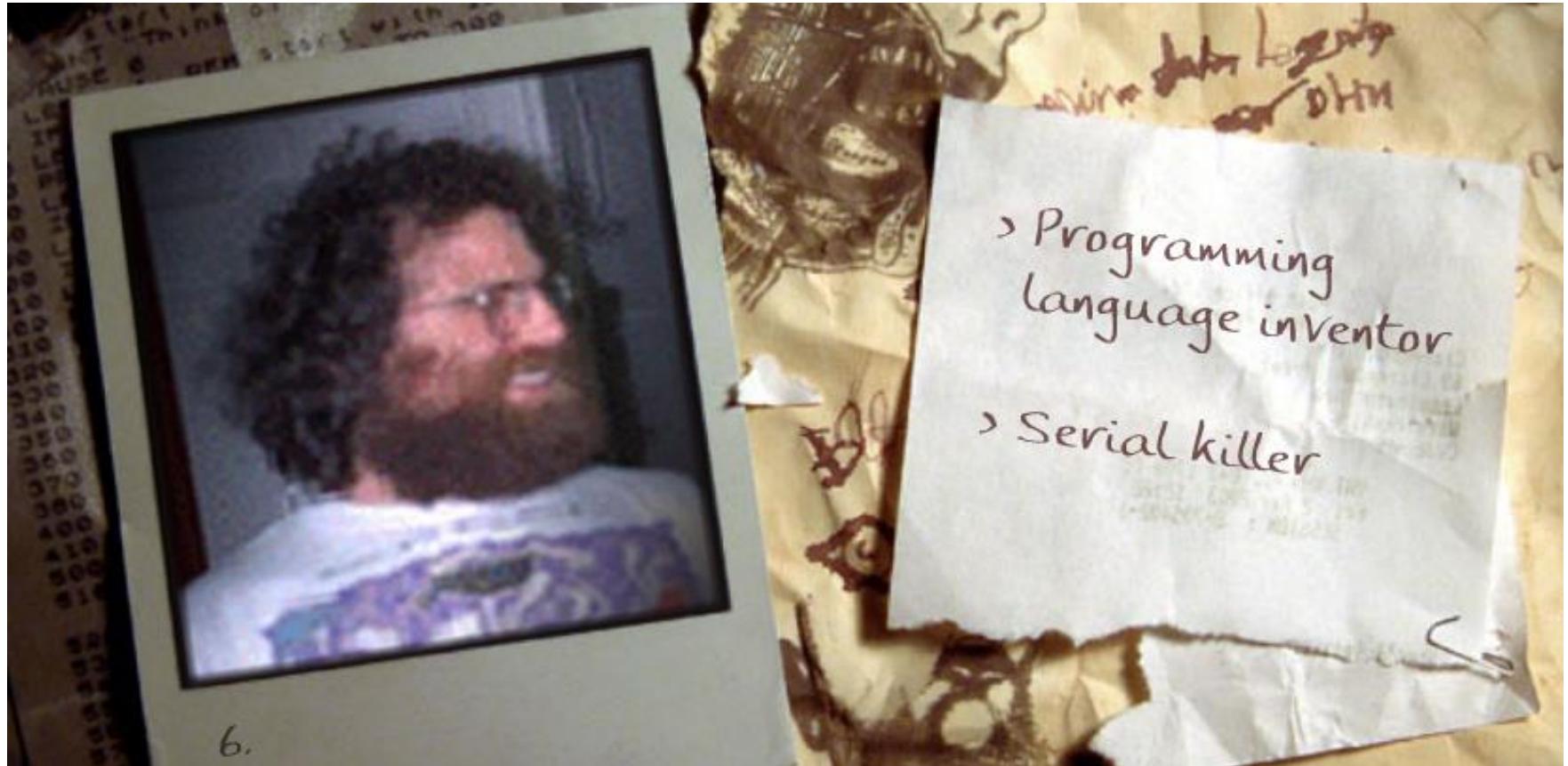
```
void foo()
{
    Gassy a, *b;
}

int main(void)
{
    int j;
    Gassy *ptr, x;

    ptr = new Gassy;
    for (j=0;j<3;j++)
    {
        Gassy a[5];
        foo();
    }
    delete ptr;
}
```

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

Can you guess?



- › Programming language inventor
- › Serial killer

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

Class Composition

Class Composition

Class Composition

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;  
};
```

Class Composition

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

myBelly

myBrain

But they're not initialized yet!

{

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  
    }  
private  
    Stomach myBelly;  
    Brain 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).

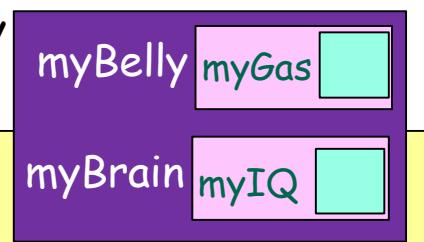
Finally, after all embedded objects have been destructed, the outer object's memory is freed.

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
    }
}
```

This implicitly added constructor...

```
void celebrate()
{
    cout << "Happy " << myAge <<
        "' th birthday!";
}
```

so they'll have a random value here!

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

won't initialize your object's scalar variables...

Just be careful!

Remember, that the auto-added default constructor...

WON'T initialize your scalar member variables!

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

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

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

```
class Bar  
{  
public:  
    Bar() { ... }  
    ~Bar() { ... }  
private:  
    Ack myAck;  
};
```

#3 #5

Call Ack()

{ ... }

Call ~Ack()

{ ... }

#4

#5

```
class Foo  
{  
public:  
    Foo() { ... }  
    ~Foo() { ... }  
private:  
    Bar myBar;  
};
```

#2 #6

Call Bar()

{ ... }

Call ~Bar()

{ ... }

#2

#3

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 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 C++

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

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

private:
    int myGas;
};
```

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

And here's how we create a new **Stomach** variable...

```
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!

Advanced Composition

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

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

```
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 Constructors

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

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

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

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

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

Class Composition

whose constructor requires parameters!

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

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

```
class HungryNerd
```

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

This is called an "initializer list".

Call myBrain's default constructor

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

```
private:
```

```
    Stomach myBelly;
    Brain myBrain;
```

```
};
```

We use it to initialize an embedded object...

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

Advanced Class Composition

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

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.

```
class HungryNerd
```

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

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

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

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

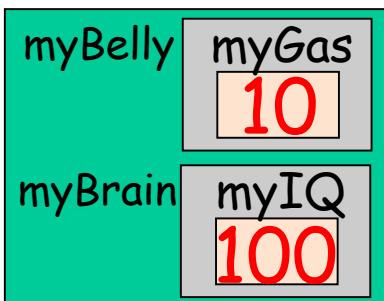
Advanced Class Compos

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

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



class HungryNerd

```
{  
public:  
    HungryNerd()  
        : myBelly(10)  
    {  
        myBelly.eat();  
        myBrain.think();  
    }  
private  
    Stomach myBelly;  
    Brain myBrain;  
};
```

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!

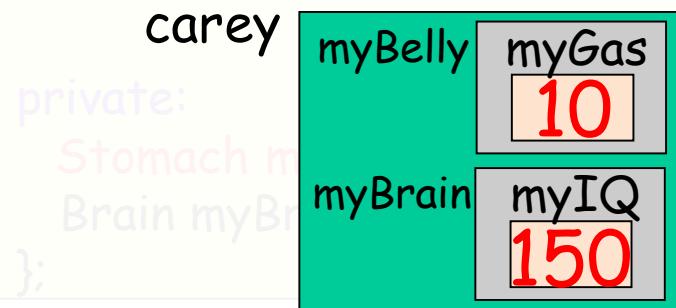
```
int main(void)  
{  
    HungryNerd carey;  
    ...  
}
```

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

You **must** always add your initializer list to your actual **constructor definition** (whether it's defined **inside** or **outside** your class).

```
class HungryNerd
{
public:

    HungryNerd()
        : myBelly(10), myBrain(150)
    {
        myBelly.eat();
        m_age = 19;
    }

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

Then you must have your
initializer list here.

```
class HungryNerd
{
public:
    HungryNerd(); // Constructor body defined outside

private:
    Belly myBelly;
    Brain myBrain;
    int myAge;

    HungryNerd::HungryNerd()
        : myBelly(10), myBrain(150)
    {
        myBelly.eat();
        m_age = 19;
    }
};
```

And **NOT** here.

So if you define the
body of your
constructor
OUTSIDE of your
class definition...

Initializer Lists

If you like, you **may** also use the initializer list to **initialize** your scalar 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 scalars if you like, but it's not required either way.

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

Instead of initializing your scalar variables **here** in your **constructor body**...

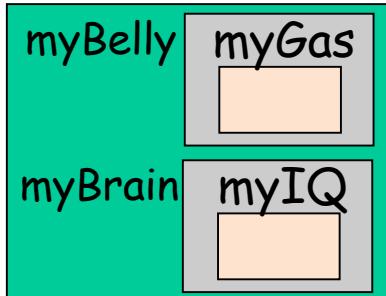
Advanced Class Composition

```
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...

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

carey



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

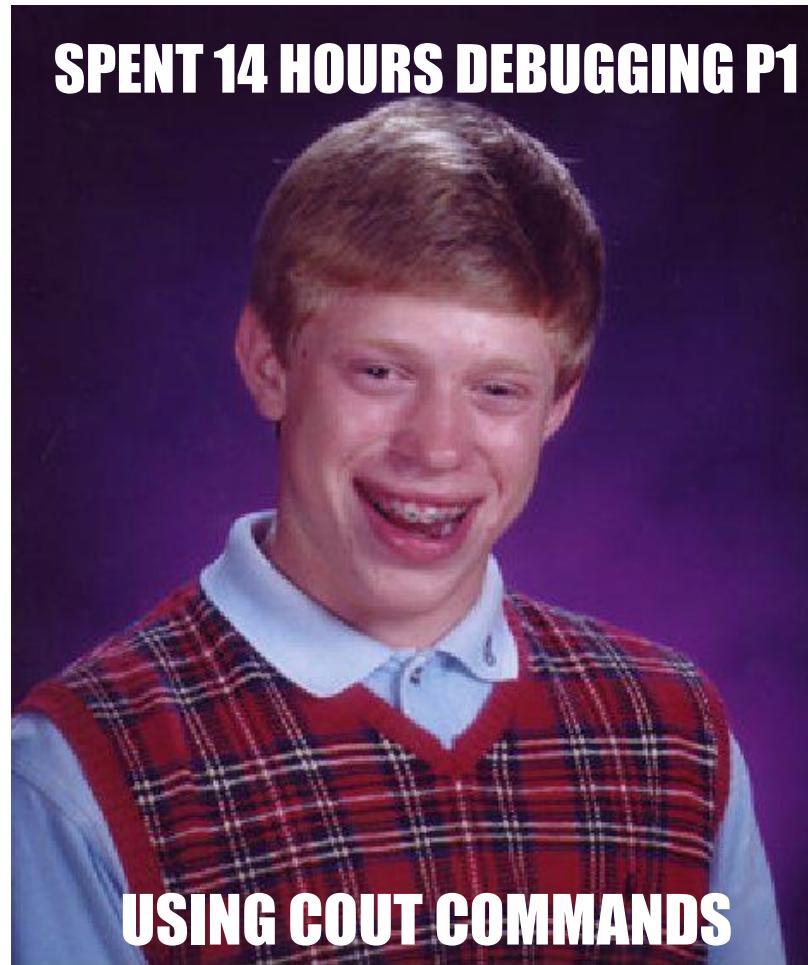
```
Stomach myBelly;
Brain myBrain;
```

We don't need to have constants here!

And these can even be more interesting expressions!

```
int main(void)
{
    HungryNerd carey(75);
    ...
}
```

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...

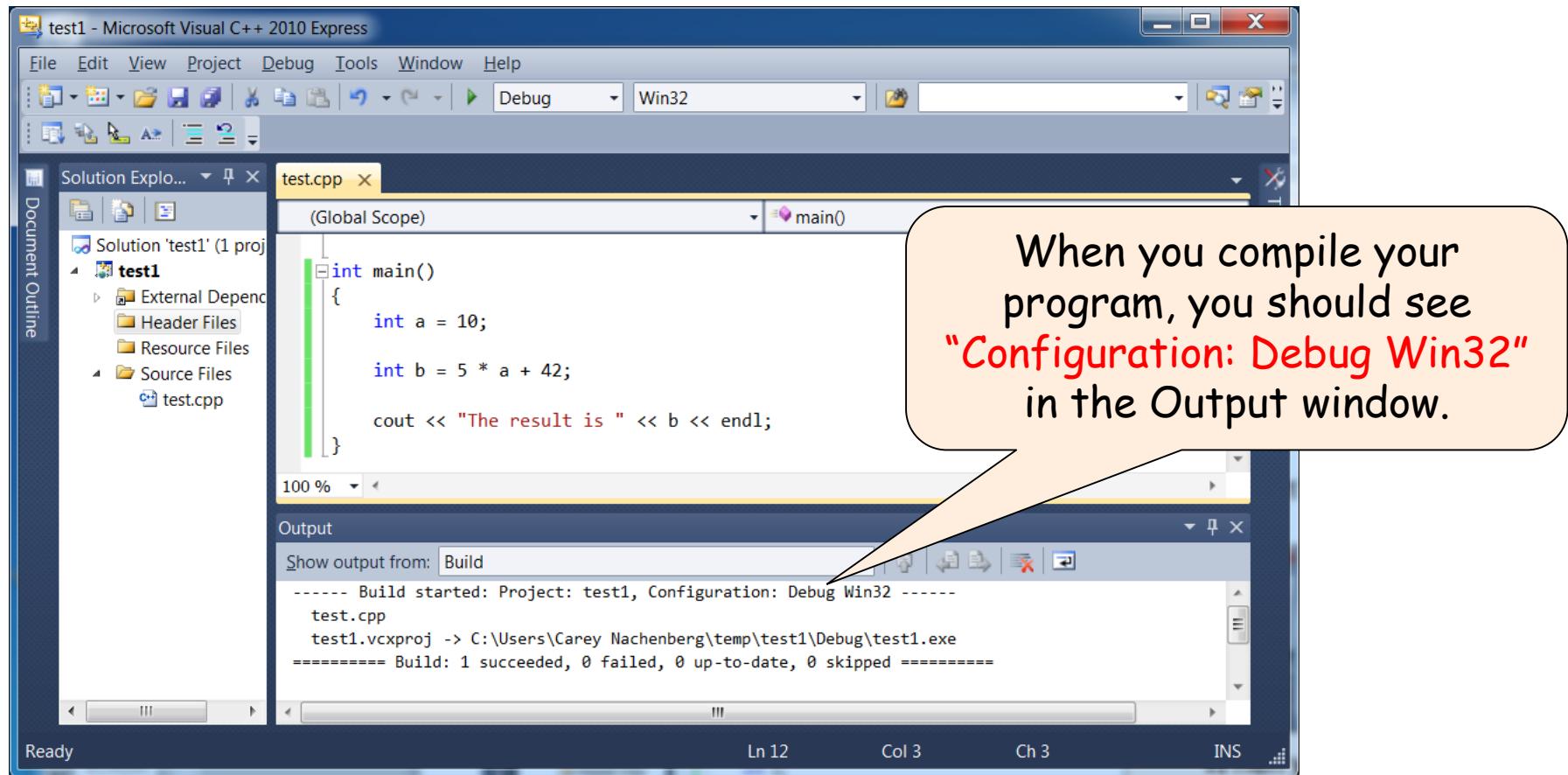
When you could find it in **30**
seconds with the **debugger**.

So pay attention!



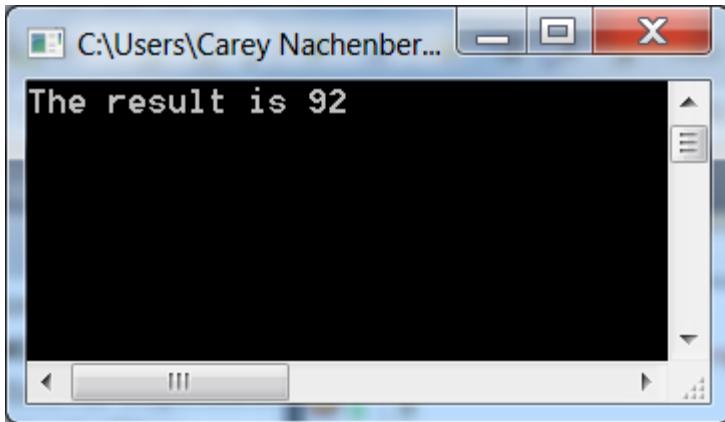
Debugging

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

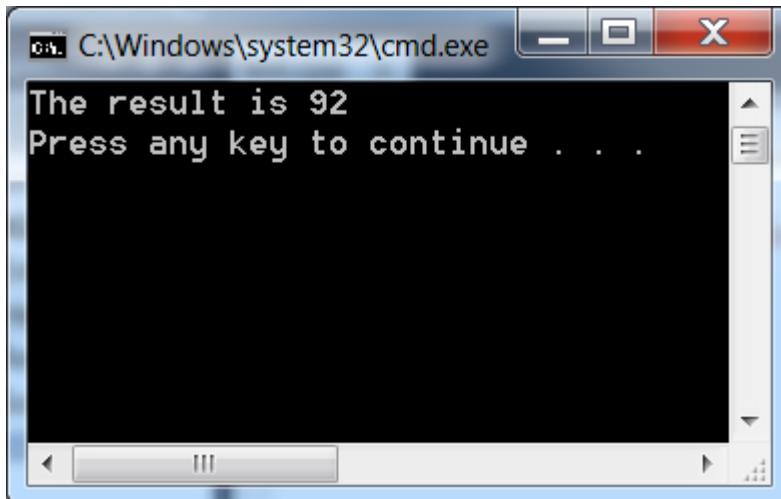


Debugging

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

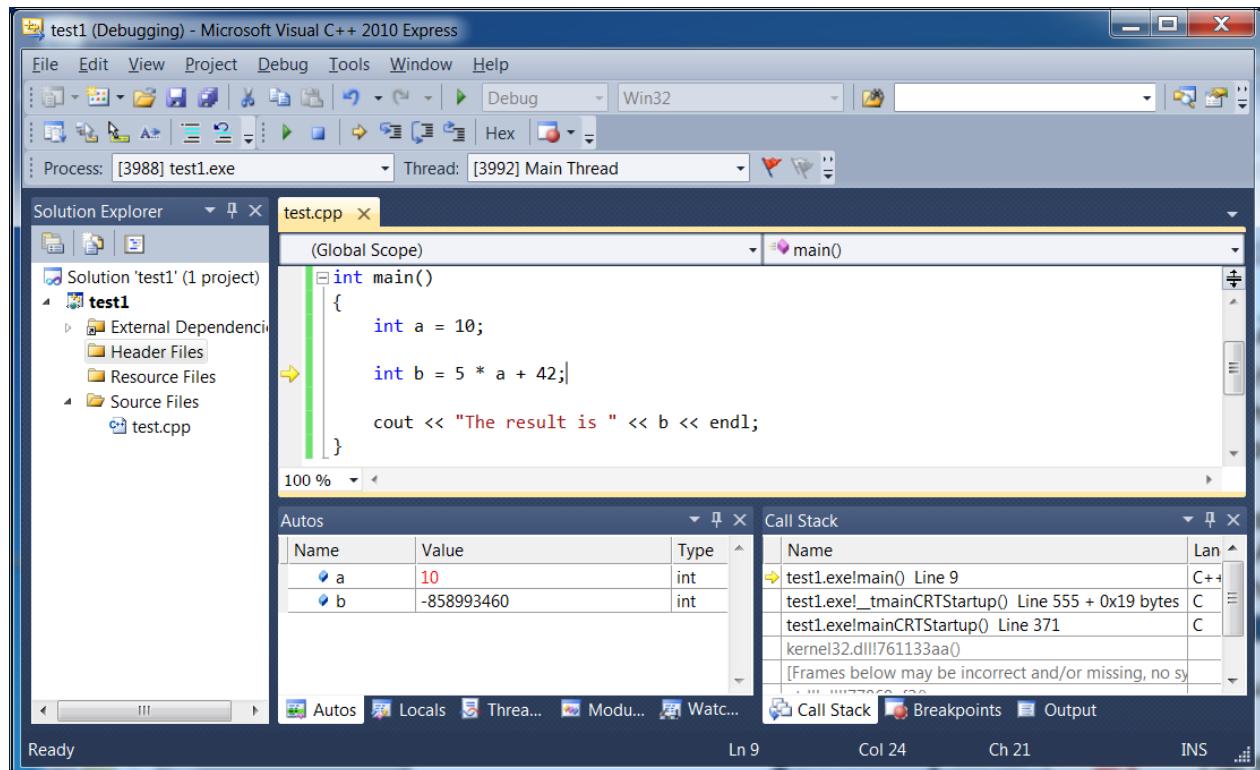
You can step through your program one line at a time using the F10 and F11 keys.

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.

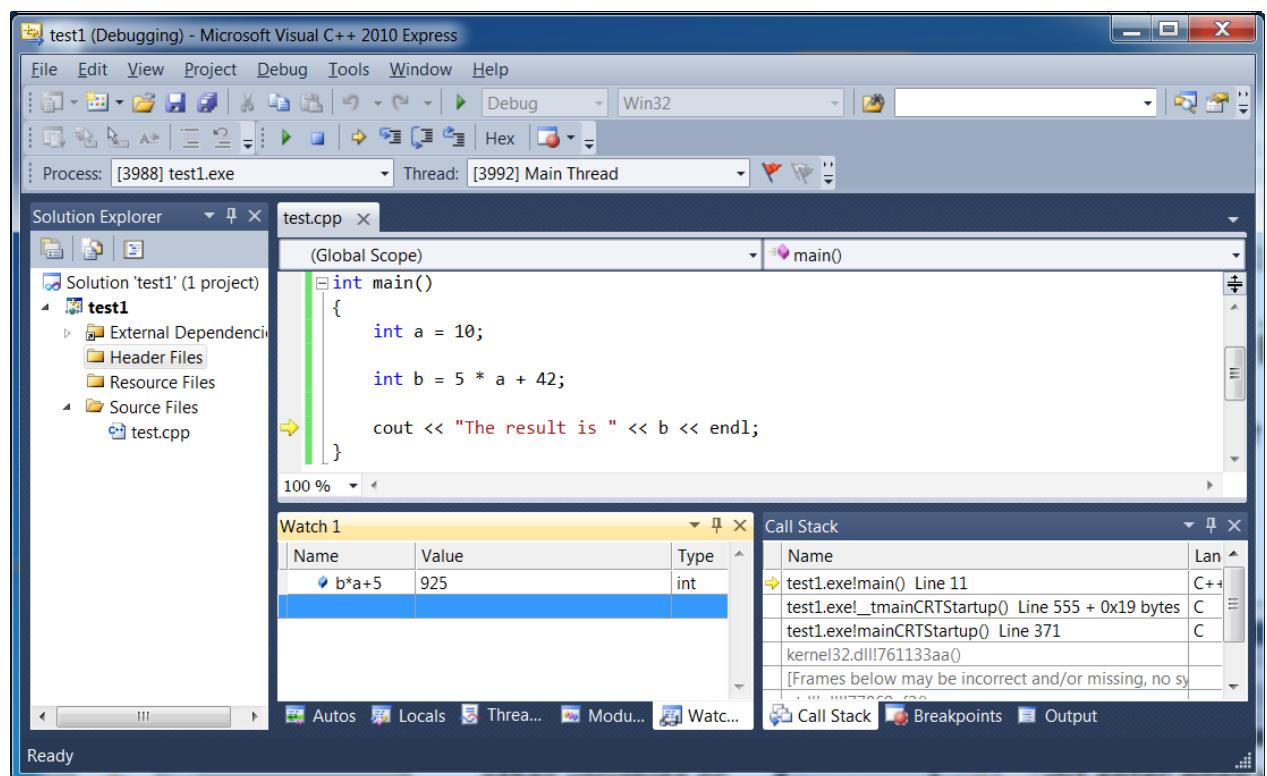


Debugging

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.

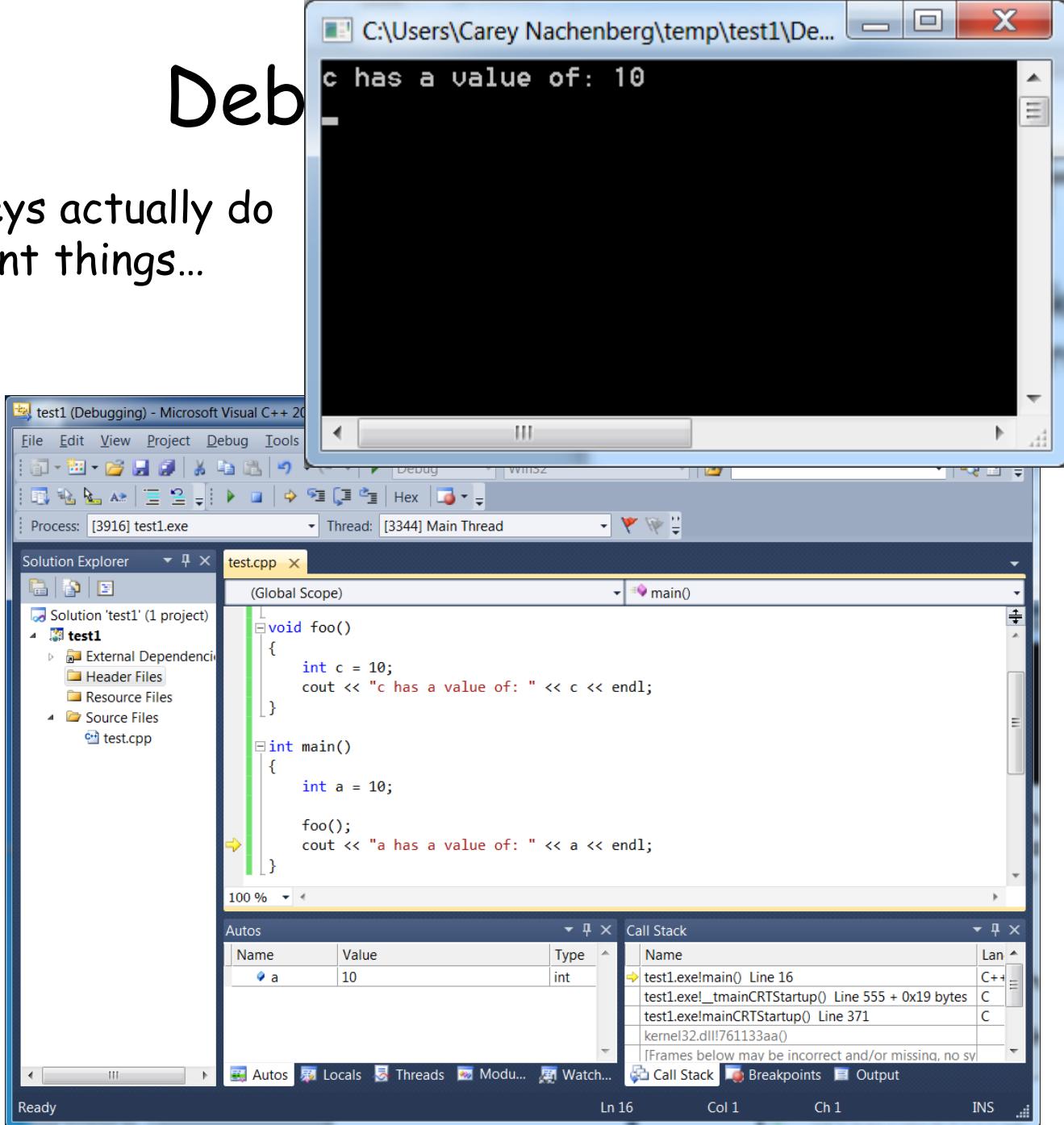


Debug

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.

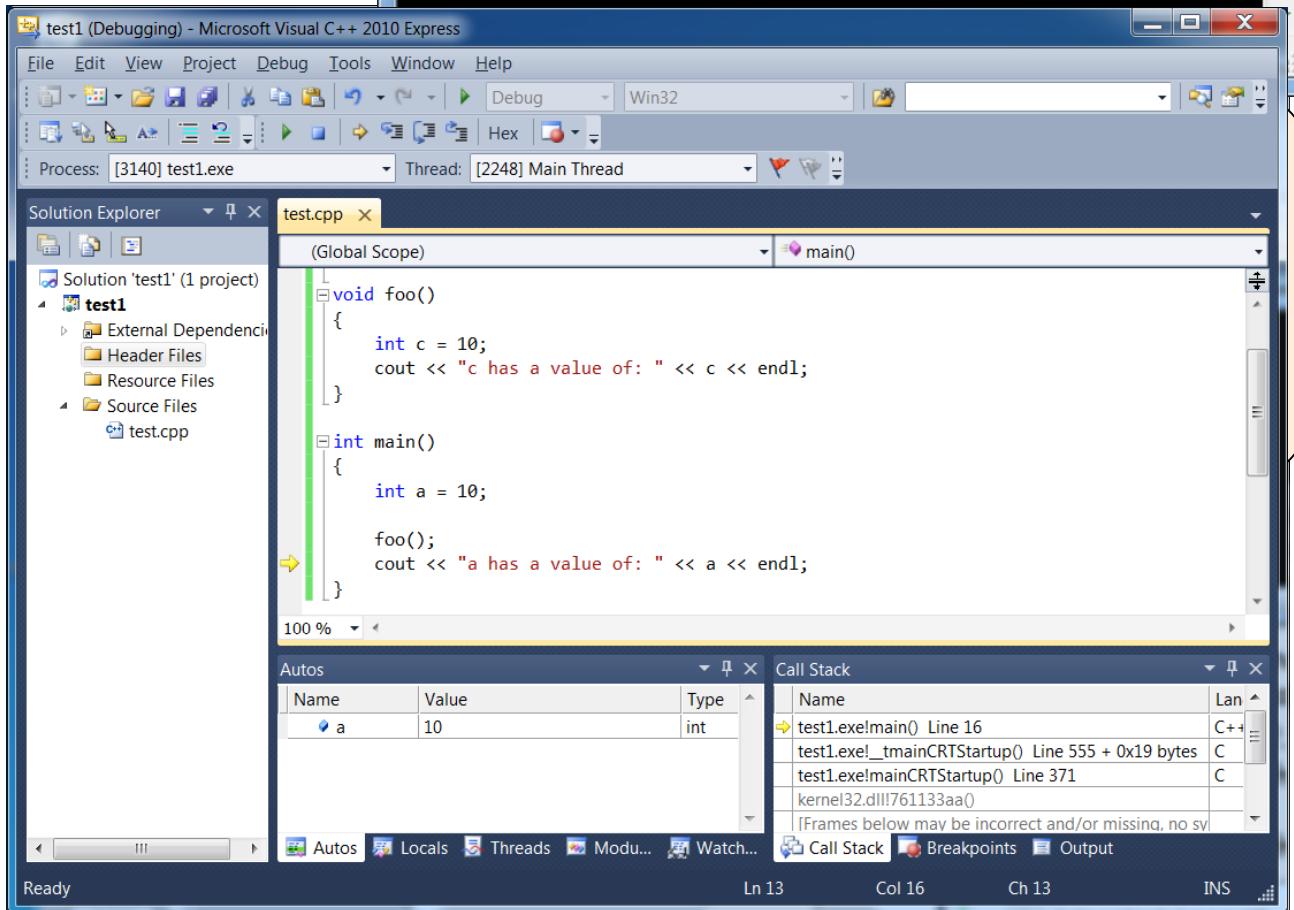
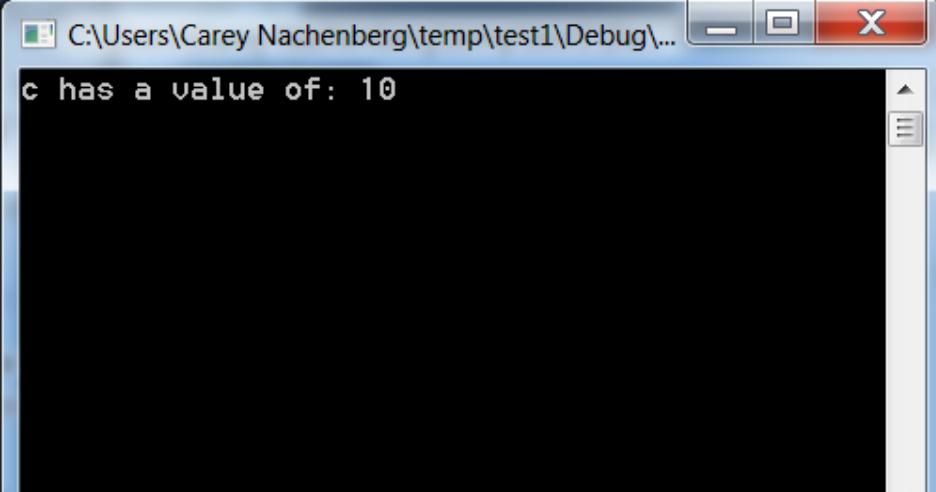
So F10 lets you quickly run a line of code without delving into the details.



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



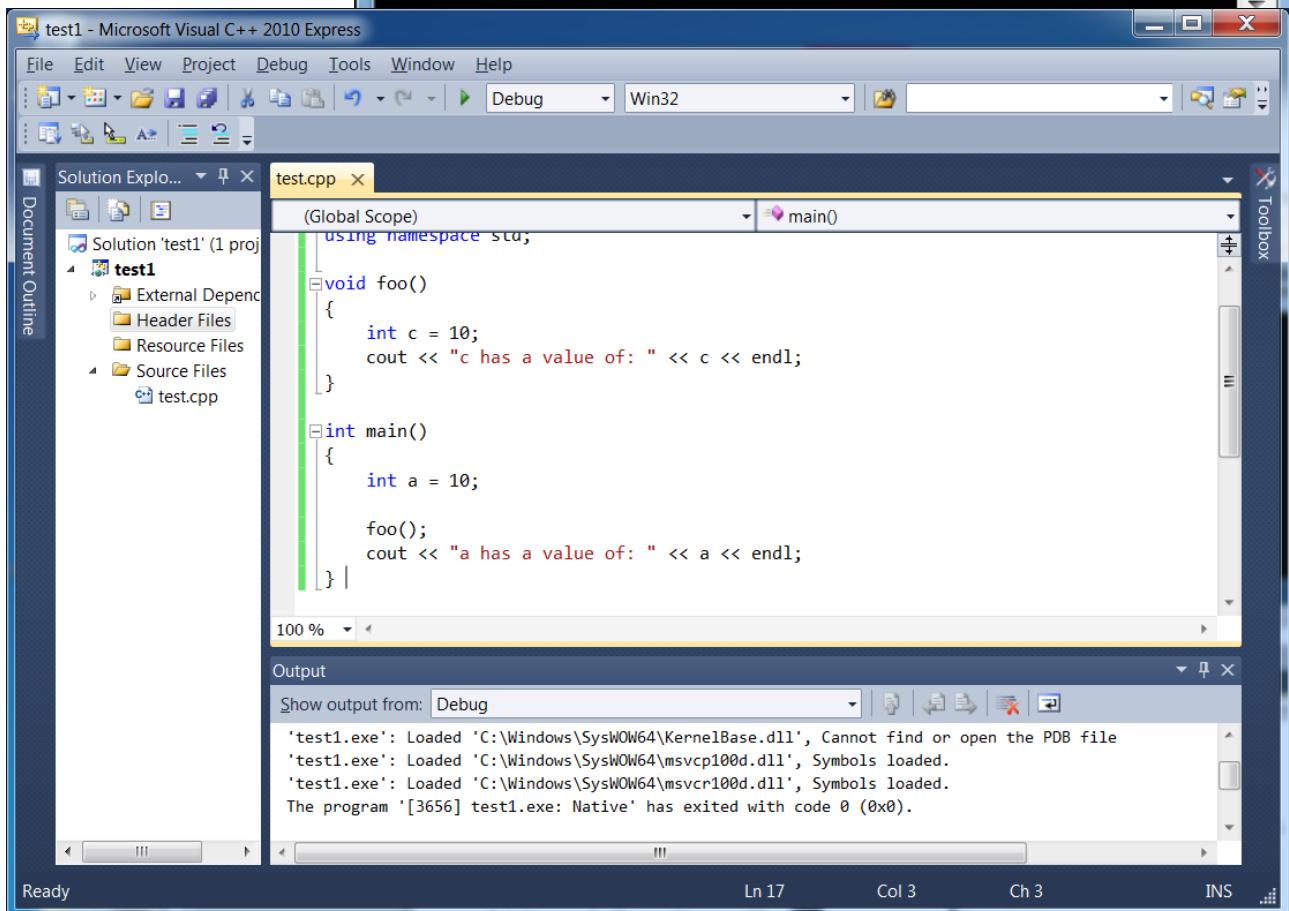
If at any time while
you're tracing
line-by-line you want
to stop tracing
line-by-line...

And simply let the
rest of your program
run normally...

Just hit F5
and Visual Studio will
run your program
until completion!

Deb

```
C:\Users\Carey Nachenberg\temp\test1\Debug\...
c has a value of: 10
a has a value of: 10
-
```



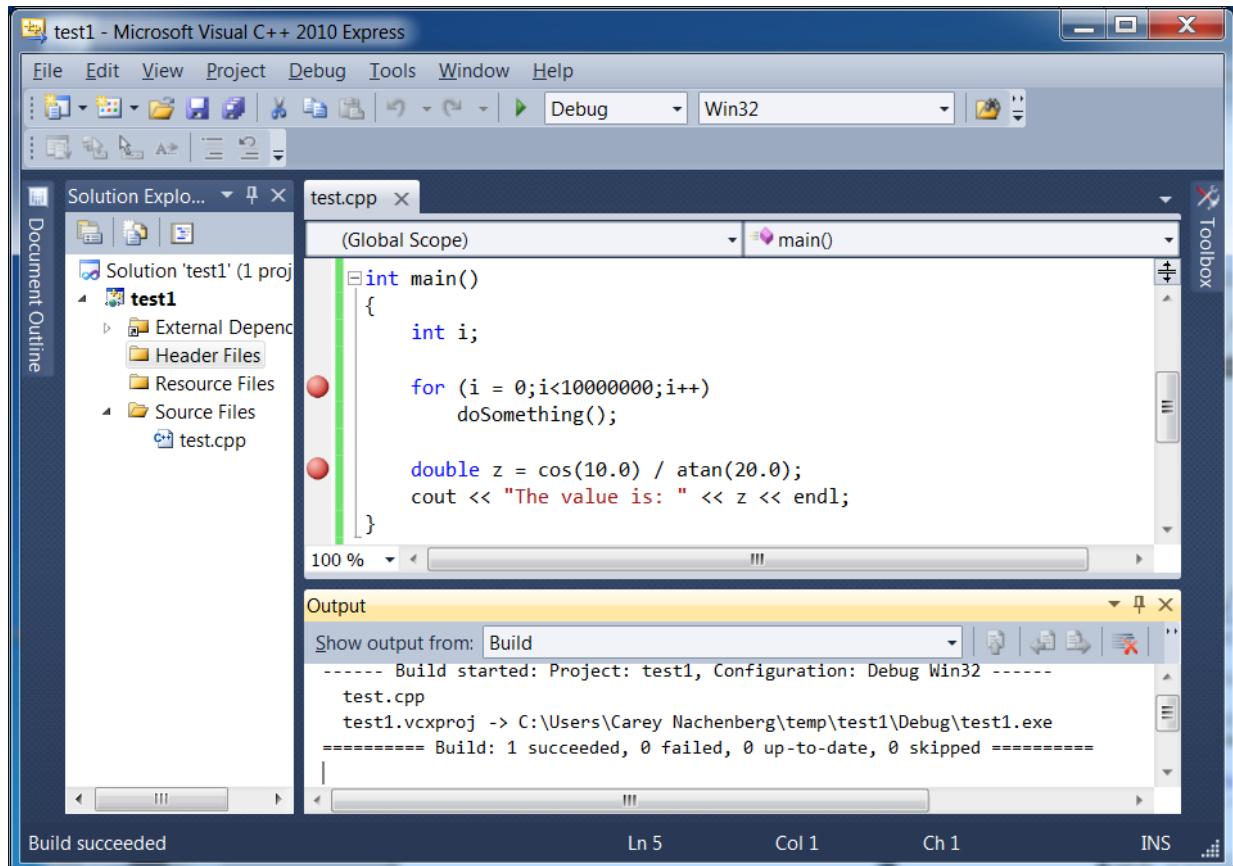
Debugging

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

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!



Class Challenge

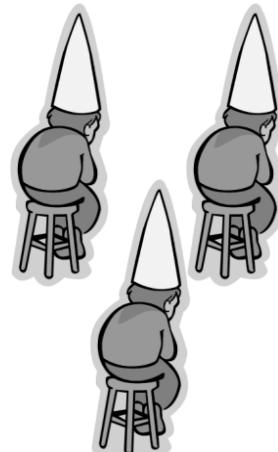
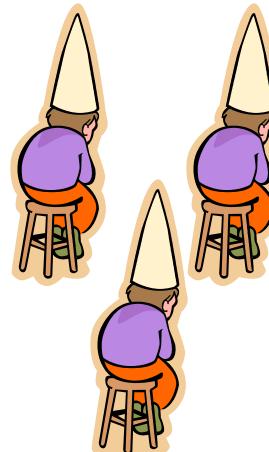
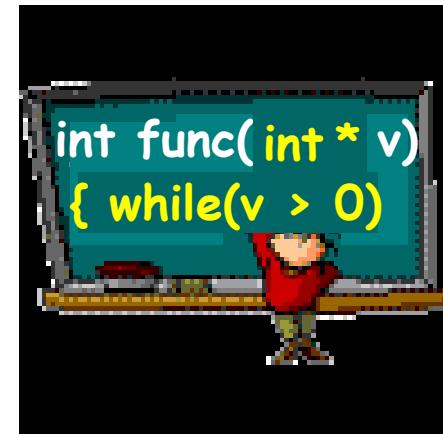
RULES

- 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.

A Few Final Topics

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



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!"
}
```

You'll get linker
errors.

Only include .H files
within a .CPP file.

file2.cpp

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

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

Topic #1: Include Etiquette

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

someHeader.h

```
#include <iostream>  
  
using namespace std;
```

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.

hello.cpp

```
#include "someHeader.h"  
  
// BUT I DON'T WANT THAT  
// NAMESPACE! YOU BASTARD!  
  
int main()  
{  
    cout << "Hello world!"  
}
```

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;
}
```

Topic #1: Include Etiquette

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

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
```

calc.h

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

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...

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();
}
```

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?

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
```

student.h

```
#ifndef STUDENT_H  
  
#define STUDENT_H  
  
#include "calc.h"  
  
class Student  
{  
public:  
    void study();  
    ...  
private:  
    Calculator myCalc;  
};  
  
#endif // for STUDENT_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!

alcohol.h

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

Last Topic: Knowing WHEN to Include .H Files

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 its .h
file first... Right?

Well, not so fast!

A.h

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

B.h

```
#include "A.h"

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

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

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!

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).

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 WHEN 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 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?

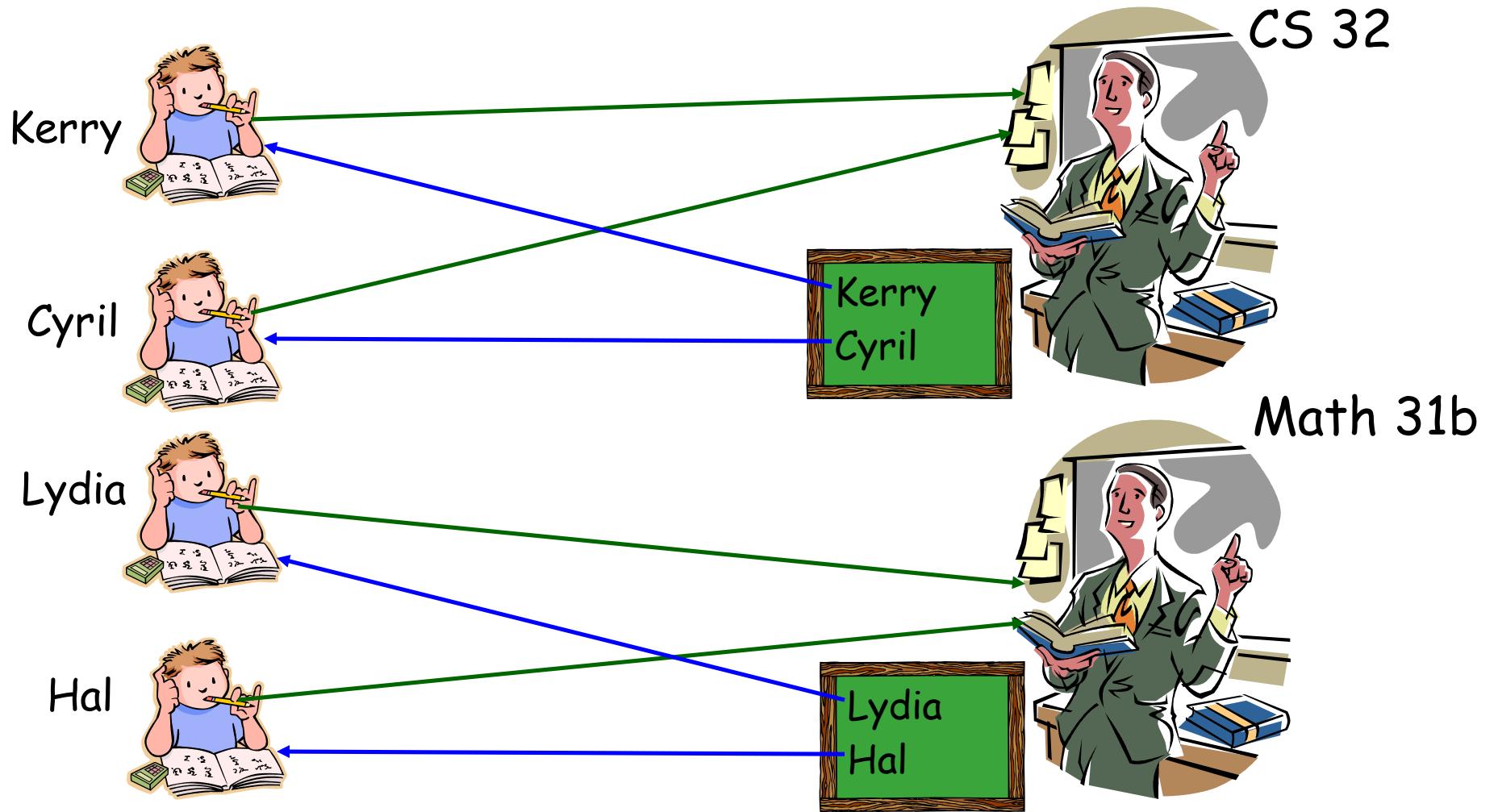
There are two reasons:

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"
```

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?

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 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**).

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?

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!"; }  
  
}
```

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.

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!

```
bool poop(string name, int numberOfWipes)
{
    cout << person << " 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!

```
bool poop(string name, int numberOfWipes=2  
{  
    cout << person << " 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!

```
bool 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)
{
    ...
}
```