Recitation 4

Topics

- Pointers, addresses, address of operator, dereferencing pointer values
- The `this` keyword
- dynamic objects
- vectors of pointersc
- one way association (implemented via pointers)
- Use of the Debugger with pointers and addresses

Recitation

Today's recitation is "task based."
Work through these Tasks on your own, making sure that you understand what is going on in each Task.

And, as always, ask us if you need help!

You will see that many of the directions specify how to use the debugger in Visual Studio. Other IDE's, such as the Mac's Xcode, should be similar, but you may have to check the menus to find the matching action. (Yes, we should update this for Xcode as more students now are using Macs.) For those using a Unix environment without an IDE, here is a very brief guide to the gdb debugger.

Task 1: [Lame] Create a new C++ program, "rec04.cpp" with the usual comment at the top, identifying you and the program. Yes, this is lame, but so many students skip this...

Task 2: Add the following code to `main()`.

```
int x;
x = 10;
cout << "x = " << x << endl;
```

Run the program. Hopefully, you know what to expect.

Task 3: Using the debug facilities of your environment, explore what is going on behind the scenes of our program.

Set a breakpoint at the beginning of main and run to the breakpoint.

If using an IDE new windows should appear. One of them is the output window of your program which should display as usual. However, it may disappear at times while the Debugger returns you to your code.

Assuming Visual C++ (other IDE's are similar) in Debug mode, your environment (if you use the default layout) displays four main windows: the code window or editor (upper left), the Solution Explorer window (upper right), the Variables window (bottom left), and the Call Stack/Breakpoint window (bottom right). Most of these windows contain tabs at the bottom for other displays in each window (for example the Autos, Locals and Watch 1 tabs in the Variables window). If you don't see a debug window, you can use the Debug menu, Windows submenu, to activate missing debug windows. If you don't see the Solution Explorer you can show it by using the View menu's Solution Explorer option.

Your program is waiting for you. In debug mode you are stepping through the execution of the program. You did "run to cursor" and now the program has stopped *before* executing the statement your cursor was on. A yellow arrow at the left of the editor/code window should be pointing to the assignment statement `x = 10;`. That is the next statement that will be executed. Select the "Locals tab" if it is not selected. There should be one variable displayed in the variables window: `x`, which should have a "garbage" value. Garbage is a technical computer science term for a value that was *not* placed into a variable under the programmer's control and must never be used under any circumstances. The value displayed may be -858993460 but this might be different on your machine since this value is just whatever bits happened to be turned on in that memory location when the program used it for `x`. The assignment of 10 to `x` has not yet been executed, so `x` contains GARBAGE.

Task 4: Execute the next statement by pressing selecting *Step Over* in your degugger (the F10 key is the shortcut in Visual Studio). The Variables window will change and display `x` with a value of 10 in red. Red indicates that a value has changed in memory. The yellow arrow will have moved to the next statement waiting to be executed.

Task 5: In the Variables window, click the Watch 1 tab. The Watch window allows the programmer to type in expressions (like the name of a variable or x*y-17) to see the value of that expression AT THIS MOMENT in the program's execution. Click under the Name column in the first box. If you

want to see the value of `x` (which means what's currently stored in `x`) you would type `x`. Do that now. You should see 10 under the Value column.

Memory can be considered to be a very large array of bytes. Each byte has an address, starting with the first address at byte 0. Now that we know that `x` contains 10 - that 10 is stored in the memory location we call `x`, then "Where is `x` in the computer's memory?" is the next question. "Where is that 10 stored?" is the same question as "where is `x`?"

Task 6: In C++ there is a "unary operator" (an operator that works on just one operand) that evaluates to where in memory something is. It's called the "address-of" operator. The address-of operator is notated as the ampersand. The expression `&x` yields the memory location where variable `x` is placed during the current execution of the program (it may be in a different location every time we run the program.) You can use the Watch window to see where `x` is. Enter `&x` in the watch under the name column. [Note that name is a pretty confusing header for the column because `&x` is not a "name." it's an expression that involves a name.] The syntax for the address-of operator requires it to be to the left of the variable (or parameter) that we want to find out the address of. The value might be 0x0012fed4 (very likely it will be different on your machine). The 0x prefix indicates that the following digits are hexadecimal (base 16). This format is standardly used for memory locations. (Review or learn hexadecimal counting if you are not familiar with it.) In C++ you can type 0xB whereever you need the number 11 (in decimal) as in `cout << 0xB;` and the number 11 (in decimal display by default) will appear in the output window.

[Note that you can ask for hexadecimal output by sending the output manipulator hex to an output stream: cout << hex << 0xB; And you'll need to use the dec manipulator to change back to decimal output: cout << dec << 0xB; Also note that this is a minor topic for this course and that there an `oct` for octal manipulator.]

You may notice that you have seen the '&' used in a different context.
What context was that?
It was part of the typename for reference parameters.
This is NOT the "reference to" operator - there is no such thing as the "reference to" operator.
& is used in two different contexts in C++ but only one of them - the address of operator is an operator.

Task 7: There should also be a small triangle with a '+' sign to the left of `&x`. Click it and it should expand to display more information and become a '-' sign. The Watch window now shows the value of `&x` and under it, the value stored AT the address of `x`. This means the 10 is stored in the memory location that is the value of `&x` - 10 is stored where `x` is - of course! Now, we will explore pointer variables. Stop the Debugger (SHIFT-F5).

Task 8: Add the following below your code to `main()`.

```
int* p;
p = &x;
cout << "p = " << p << endl;
```

Run your program (Ctrl-F5). Does the value stored in p match the value you saw in the Watch window for the address of ;`x`? It should. What this code does is take the address of x (i.e., `&x`) and copy that value into `p` (using `p =`). That's the way a normal assignment statement works: evaluate the right hand side and copy that value into the left hand side. The only thing that might be new to you (if you've been asleep in lecture) is the *type* of `p`. Is `p` an `int`? **No**. Does it mean "multiply `int` times `p`"? No, of course not. This is a type you may not have seen before: *pointer to int*. The asterisk (`*`) in a type name means "pointer to" and all pointers must be pointers to some other type. Here it's a pointer to an `int`.

The type "pointer to int" can hold the address of an `int` variable or parameter. So you can think of `p` as being able to hold the address of an `int`. It can hold where an `int` can be stored. We say that "*p points to x*" when `p` contains where `x` is. [Note that there is not a type called "pointer" - it must be "pointer to ___" where ___ is some typename.]

Place the cursor on the line `int* p`; and do Run To Cursor (right click and choose).

Look in the Locals window. What are the types of `x` and `p`? You should see `int` for `x` and `int*` for `p`. What is the value of p? Garbage because it has not been initialized, assigned to or read into. Look in the Watch window. What are the types of x and &x? You should see `int` for `x` and `int*` for `&x`.

So the type of the expression `&x` is `int*` which is the type you used when you defined `p`. So it's fine to place the value `&x` (where `x` is) into the variable `p` to make `p` point to `x` (to make `p` contain the address of `x`).

Hit F10 (Step Over) and look in the Locals window. What value was stored in `p`?

`p` really is just a variable, just as `x` is a variable. We can assign to it. Let's try something to emphasize the type differences between the type `int` and the type `int*` ("pointer to int").

Task 9: Try to assign the value for `&x` to `p` directly:

```
p = 0x0012fed4; // using the value from my Watch window
```

What happens?
Read the error message. It should be something similar to:

```
error C2440: '=' : cannot convert from 'int' to 'int *'
```

That's pretty clear for an error message for .NET!

0x0012fed4 is NOT an `int*` value. It's an `unsigned int` most likely. Try just a plain old decimal integer value like 100000 and you will get the same error message.

Note that C++ is VERY STRICT about pointer types.

[We don't cover casting an integral value to a pointer to an int here - ask someone outside of lab.]

Note that the '*' can be written several ways when indicating a pointer type and the compiler sees as all being exactly the same:

```
int* p;     // it's very clear that the * is part of the type's name – the type is "pointer to int"
            // PREFERRED
int *p;     // it looks the * is part of the variable name but the compiler reads it as above
int * p;    // it looks more like multiplication but the compiler again reads it as above
            // (probably NOT a good idea to use)
int* p1, p2, *p3;   // This is compile-able code but's it's VERY unclear as the type of p2.
                    // Since the * is part of the type's name, it's clear that p1 is an int*; and it's pretty clear that
```

```
                    //  p3 is an int* but notice that it
looks like p2 is also an int* – but it's not!
                    // (definitely NOT a good idea to mix
int and int* definitions
```

Like the '&' symbol, the '*' symbol is overloaded in C++ for two operators...
* (binary context) arithmetic multiplication operator
* (unary context) deference operator (See Task 10)
... and is also used as part of a typename: `T*` (the '*' here is NOT an operator).

Task 10:

Comment out `p = 0x0012fed4;`
and Run To Cursor on

```
cout << "p = " << p << endl;
```

Look in the Locals window at `p`.
There is a '+' sign just as there was in the Watch window for `&x`.
Expand the '+'. What's there? 10
That's what we saw when we expanded the '+' on the expression `&x` in the Watch window.
10 is the value stored in `x` and `p` contains where `x` is. `p` points to `x`.
Expanding the '+' tells the system to "*go to where this pointer value points*".
Since `p` points to `x` (because it contains the address of `x`), we are looking at `x` when we follow `p` to where it points.
This is called "*dereferencing a pointer*" in C++.

In C++ there is a unary operator for doing this dereferencing. It can only be used on something that is or contains an address - the pointer.`p` is of type `int*` so it can contain the address of an `int` so we can derefernce it and get the `int` value it points at.

Add these statements after your code in `main()`.

```
cout << "p points to where " << *p << " is stored\n";
cout << "*p contains " << *p << endl;
```

The expression `*p` is the *dereference operator* applied to the variable `p`. This tells the computer to follow the pointer value stored in `p` (the address stored in `p`) and get the value stored where `p` points. The 10 we stored in `x` is what `*p` is referring to. `*p` is a memory location just like `x` is.

`*p` and `x` refer to the same memory because we did `p = &x;`

Task 11: Can we change x using p?

Add this statement after your code in `main()`.

```
*p = -2763;
cout << "p points to where " << *p << " is      stored\n";
cout << "*p now contains " << *p << endl;
cout << "x now contains " << x << endl;
```

Place the cursor on `*p = -2763;` and Run To Cursor.
Look at `x` and `p` (make sure the '+' is expanded so you can see the value `p` points to).
Both 10's are red because when 10 was assigned to `x`, it changed the memory that `p` points to.

Hit F10 (Step Over) and watch `x` and `p` and the expanded `p`.

There is now a red -2763 in both places in the Locals window because both places refer to the same memory location.

Task 12: Add the following code

```
int y(13);
cout << "y contains " << y << endl;
p = &y;
cout << "p now points to where " << *p << " is stored\n";
cout << "*p now contains " << *p << endl;
*p = 980;
cout << "p points to where " << *p << " is stored\n";
cout << "*p now contains " << *p << endl;
cout << "y now contains " << y << endl;
```

What's going on? Since `p` is just a variable, we can assign to it as long as what we assign is a pointer value.
We made `p` point to `y` and changed y through `p`.

Task 13: Add the following code

```
int* q;
q = p;
cout << "q points to where " << *q << " is stored\n";
cout << "*q contains " << *q << endl;
```

Look at all the ways to get to the place we call `y`. (Expand the '+' for `q`.)

As long as the types are the same, pointer values can be assigned. We say that the above code makes `p` and `q` point to the same place (`y`).

Task 14: We said earlier that C++ is very strict on types when dealing with pointers.
Let's explore.
Define double variable named `d` and initialize it to 33.44.
Define a variable named `pD` that can contain a pointer to a `double` and initialize it so that it points to `d`.

```
double d(33.44);
double* pD(&d);
*pD = *p;
*pD = 73.2343;
*p  = *pD;
*q  = *p;
pD  = p;
p   = pD;
```

What lines are the compilation errors on?
The VERY STRICT type rule for pointers is that there is NO COERCION between any pointer types.
An `int*` cannot be cast to a `double*` even though an `int` can be cast to a `double`.
A `double*` cannot be cast to an `int*` even though a `double` can be cast to an `int` (with truncation).
Similarly with `int*` cannot be assigned to `double*`.

Comment out the error lines and step through this code looking at the Locals window.

Task 15: What about const pointers and their values and access?
Copy this code in your program and uncomment the commented lines one at a time to see what's allowed and what's not.
Note that there are four ways to deal with const-ness when dealing with pointers.

```
int joe( 24 );
const int sal( 19 );
int*  pI;
//  pI = &joe;
// *pI = 234;
//  pI = &sal;
// *pI = 7623;
```

```
//const int* pcI;
//  pcI = &joe;
// *pcI = 234;
//  pcI = &sal;
// *pcI = 7623;

//int* const cpI;
//int* const cpI(&joe);
//int* const cpI(&sal);
//  cpI = &joe;
// *cpI = 234;
//  cpI = &sal;
// *cpI = 7623;

//const int* const cpcI;
//const int* const cpcI(&joe);
//const int* const cpcI(&sal);
//  cpcI = &joe;  // *cpcI = 234;
//  cpcI = &sal;
// *cpcI = 7623;
```

Task 16:

The syntax for defining a variable is:
TYPE variableName;

TYPE can be *any* type.

For pointer types, '*' is part of the type's name: "TYPE*" is a "pointer to TYPE" type.

```
TYPE* variableName; // can hold a pointer to TYPE value
```

If you've defined a struct, isn't that a type?
So it follows that we can have pointers to structs.

Add this struct definition to your code.

```
struct Complex {
   double real;
   double img;
};
```

and define a `Complex` type variable named `c` initialized to the complex value (11.23,45.67)
and define a variable named `pC` initialized to point to `c`.

```
Complex c = {11.23,45.67};
Complex* pC(&c);
```

How do you access the fields in this variable? Try writing them on the screen.
When you deference a pointer to a type you get a thing of the type that the pointer points to.
`*p` is of type `int`. `*pD` is of type `double`. So when you dereference a pointer to a `struct Complex`, you get a `Complex` struct. So try this:

```
    cout << "real: " << *pC.real << "\nimaginary: " << 
*pC.img << endl;
```

What's wrong? The precedence of operators is giving you a problem. The operator `*` (dereference) has a LOWER precedence than the operator. (member access) so the solution is parentheses.

Don't simply copy and paste the next code into your .NET IDE.

Try typing it and watch what the IDE does when you get to the dot member access operator (`.`) in `(*pC).`
The IDE shows you what can follow the dot because the type of `(*pC)` is `struct Complex`.

```
    cout << "real: " << (*pC).real << "\nimaginary: " << 
(*pC).img << endl;
```

The notation `(*pC).` is a bit clumsy and as often is the case in C++, its designers added an operator specifically for this purpose. This operator is written `->` and often pronounced "arrow" but really means "dereference operator for pointer to struct or class" which is really too long to pronounce. The `->` operator ONLY works on pointer to struct (or class) types. It must have a struct (or class) object as its left hand operand and the name of a member of that struct (or class) as its right hand operand.

Now try the ***correct*** way to deference a pointer to a `Complex` object:

```
    cout << "real: " << pC->real << "\nimaginary: " << pC-
>img << endl;
```

In the Watch window type the expression `pC->real` to see that this really is an expression that evaluates to what is stored in the `real` field of the object that `pC` points at.

Now that you know the correct operator to use to deference to the inside of a struct (or class), will you ever write code like this `(*pC).real` again? (The answer is no!)

Task 17:

What about pointers to class objects?
Copy this code into your program to see how this works.

```
class PlainOldClass {
public:
   PlainOldClass() : x(-72) {}
   int getX() const { return x; }
   void setX( int val )  { x = val; }
private:
   int x;
};

PlainOldClass poc;
PlainOldClass* ppoc( &poc );
cout << ppoc->getX() << endl;
ppoc->setX( 2837 );
cout << ppoc->getX() << endl;
```

Task 18:

Consider if you decided that a good name for the parameter in the mutator was `x`.
How will this mess things up?
Think about changing your code in this way:

```
class PlainOldClass {
public:
   PlainOldClass() : x(-72) {}
   int getX() const { return x; }
   void setX( int x )  { x = x; } // HMMMM???
private:
   int x;
};
```

Type this code in and see in the debugger if the correct assignment happens.
There needs to be a way to talk about the actual object being worked on while a method in that object is running.
The "`this`" parameter is how to indicate which object is being worked on

by a method..
It's as if there were actually a "hidden" parameter to each method of type `ClassName*` that the system initializes to be the address of the object that method is invoked on.

It might look like this:

```
int getX( const PlainOldClass* const this = &poc ) const
{ return x; }
void setX( PlainOldClass* const this = &poc, int x )  { x =
x; } // Still HMMMM???
```

We can't write code like that, but this is the concept.
Now using the "`this`" pointer/parameter will make it clear which `x` is which:

```
   void setX( int x )  { this->x = x; } // No confusion!
```

Before adding the dereferencing of parameter `this`, you were assigning the parameter `x` to itself.
Type the new and improved mutator code into your program and see that it works..

Task 19: Let's get dynamic.

Everything we have done to this point has dealt with variables that were named and created in our program.
That means the memory for them is grabbed in RAM when the program is loaded and where these kinds of variables are in RAM cannot change while the program is running.

Sometimes a program might need to have more memory than the defined variables and parameters written in the code.
Operating systems reserve some RAM to given to programs for this purpose.
This area of RAM is called the *heap* (another name is *dynamic memory*) (yet another name is the *free store*).
This is called dynamic memory (as opposed to "static" memory for variables and parameters).
In C++ the way to request more memory from the operating system is the operator: *`new`*.
The `new` expression yields a pointer value that points to the heap memory the system lets the program use.
There's only a finite quantity of RAM in the heap so a good programmer

should release any heap memory back to the OS so it can reuse it.
In C++ the operator `delete` does this.

When requesting dynamic memory, we must specify the type we want.
Whether or not to initialize that memory location depends on the problem being solved.
The initialization cannot be done using the '=' symbol as you have seen used for normal variable initializations..
It must be done using parentheses.

Add the following code.

```
int* pDyn = new int(3); // p points to an int initialized to 3 on the heap
*pDyn = 17;
```

Run to cursor on the definition with initialization. (The first of these two lines is an initialization, not an assignment.)

Expand the '+' on `pDyn`. That's garbage.
Hit F10 to do the initialization.
The value stored is an address on the heap.
Hit F10 to do the assignment.
There's the 17 on the heap, pointed to by `pDyn`.

Where is the heap?
Check the address where this 17 is:

```
cout << "The " << *pDyn
     << " is stored at address " << pDyn
     << " which is in the heap\n";
```

Notice that the value stored in `pDyn` is shown on the screen.
The output system displays pointer values in hexadecimal format by default (you do not need to use the `hex` manipulator).

Task 20: Now let's release this dynamically allocated int back to the OS. But first, let's look at what's in `pDyn`. And then look again after we do the delete.

```
cout << pDyn << endl;
delete pDyn;
cout << pDyn << endl;
```

The OS can now reuse the space where that `int` 17 was.
Notice however that `pDyn` is still holding the address of that `int`!
`delete` does not set the pointer variable to `nullptr`! (which means "not

pointing anywhere" and is discussed in the next Task).
You SEE that `delete` does NOTHING to the pointer variable.
The heap space it points to is released (`delete`d).

Our program *should not* access that space since we gave it back to the OS. But can we?

```
cout << "The 17 might STILL be stored at address " << 
pDyn<< " even though we deleted pDyn\n";
cout << "But can we dereference pDyn?  We can try.  This 
might crash... " << *pDyn << ".  Still here?\n";
```

The system may have already reused that space on the heap!
A good thinking programmer would not want to reuse memory she has released back to the OS but C++ does nothing to enforce this.
The issue here is that the `delete` operator does NOT change the contents of the pointer variable.
It's our job as good programmers to put some value into that variable that indicates that it no longer validly points anywhere.
That's what we will cover in the next Task.

Task 21: `nullptr`

C++ has the value `nullptr` as a built-in name for a special integer value with special meaning.
`nullptr` *means* not pointing to anything.
It's actually the integer value zero.
So to indicate that we no longer have a valid pointer, we need to set `nullptr` as the value:

after this statement: `delete pDyn;` we would normally write:

```
pDyn = nullptr;
```

to indicate that we deleted the pointer.

We will often initialize a pointer variable to `nullptr` so that we are can more easily find illegal dereferencing errors later when used.

```
double* pDouble( nullptr );
```

Add these lines to your code.

Task 22: dereferencing `nullptr`

Setting `nullptr`is good because it's dereferencing a `nullptr`pointer will generate an error.

Save your program before executing the next statement.
Your program is going to crash.
.NET will handle this error in different ways, depending on which version of .NET and which version of Windows you are running.
(Your program doesn't actually crash - it just notices that it would have crashed and asks you about this - but if you run this code by double clicking the .exe file, you will not be in a protected environment with a safety net built in and it ***will*** crash)

```
cout << "Can we dereference nullptr?  " << *pDyn << endl;
cout << "Can we dereference nullptr?  " << *pDouble << endl;
```

Comment out those lines of code after you've seen them crash your program.

Task 24: Deep thinking

Think about that "`this`" pointer - can it ever be `nullptr` (contain the value `nullptr`)?
DUH!!! It's the address of the object that a method is invoked on.
How can it ever be not pointing to something?

Task 24: deleting `nullptr`

It's not an error to delete a pointer that contains `nullptr`:

```
double* pTest = new double( 12 );
delete pTest;
pTest = nullptr;
delete pTest; // safe
```

Task 25: however...

It is an error to delete a pointer variable or parameter that has already been deleted.
After the delete, your program has no right to deal with that pointer value - it was returned to the system.

```
short* pShrt = new short( 5 );
delete pShrt;
delete pShrt;
```

However, how different compilers deal with this varies greatly.
What does MS .NET do?

Comment out the second delete and see if things get better.

Task 26: What about named memory locations - can they be deleted? Try these.

```
long jumper( 12238743 );
delete jumper;
long* ptrTolong( &jumper );
delete ptrTolong;
Complex cmplx;
delete cmplx;
```

No, delete can only be used with heap (or free store) (or dynamically allocated) memory - unnamed memory.

Task 27: Vectors of pointers to dynamic objects

```
vector<Complex*> complV; // can hold pointers to Complex
objects
Complex* tmpPCmplx;      // space for a Complex pointer
for ( size_t ndx = 0 ; ndx < 3 ; ++ndx ) {
   tmpPCmplx = new Complex; // create a new Complex object
on the heap
   tmpPCmplx->real = ndx;   // set real and img to be the
   tmpPCmplx->img  = ndx;   // value of the current ndx
   complV.push_back( tmpPCmplx ); // store the ADDRESS of
the object in a vector or pointer to Complex
}
// display the objects using the pointers stored in the
vector
for ( size_t ndx = 0 ; ndx < 3 ; ++ndx ) {
    cout << complV[ ndx ]->real << endl;
    cout << complV[ ndx ]->img  << endl;
}
// release the heap space pointed at by the pointers in the
vector
for ( size_t ndx = 0 ; ndx < 3 ; ++ndx ) {
    delete complV[ ndx ];
}
// clear the vector
complV.clear();
```

Note carefully that `.clear()` does NOT issue a `delete` on the things inside the vector.
The programmer must traverse the vector

and `delete` each `Complex` object stored on the heap, by deleting pointer value stored in the vector.

Task 28: Now consider reading a file of data and storing it on the heap, making the data accessible from a vector of pointers:

```
class Colour {
public:
    Colour( const string& name, unsigned r, unsigned g,
unsigned b )
        : name(name), r(r), g(g), b(b) {}
    void display() const {
        cout << name << " (RBG: " << r << "," << g<< "," <<
b << ")";
    }
private:
    string   name;    // what we call this colour
    unsigned r, g, b; // red/green/blue values for
displaying
};

vector< Colour* > colours;
string inputName;
unsigned inputR, inputG, inputB;

// fill vector with Colours from the file
// this could be from an actual file but here we are using
the keyboard instead of a file
(so we don't have to create an actual file)
// do you remember the keystroke combination to indicate
"end of file" at the keyboard?
// somehow the while's test has to fail - from keyboard
input
while ( cin >> inputName >> inputR >> inputG >> inputB ) {
    colours.push_back( new Colour(inputName, inputR,
inputG, inputB) );
}

// display all the Colours in the vector:
for ( size_t ndx = 0; ndx < colours.size(); ++ndx ) {
    colours[ndx]->display();
    cout << endl;
}
```

Here are some colours with their RGB values for testing:
Black 255 255 255
White 0 0 0
Azure 10 110 237
DeepPurple 240 72 261

(your last typing would be the keystroke combination that indicates the end of file situation)

Task 29: One-way association - implemented using the pointer type.
When we buy or build a high-end amp, it is not connected to any speaker system which would be bought separately. (None of those all-in-one cheap systems for good programmers!)
We would like to be able to model this relationship of connecting speaker systems to amps.
This relationship is a one-way association.
The speaker system does not need to know anything about the amp.
The amp does not need to know which speakers to send it's audio output to but it does need to know *which* system.
Note the use of a pointer data member, pointing to the attached speakers and the use of the `->` dereference operator to allow the amp to send a signal to the speaker system to be interpretted as a vibration of it's cone - thus making sound.
The speaker does not need to know which amp sent it's audio signal.
The amp might need to know if it is connected or not to a speaker system before sending an audio signal to it.

It might be nice to know if a speaker system is already connected when attempting to connect another system.
How to test?
Recall that the actual value - though you should never use it's value directly - is the `int` (or `unsigned`) value 0 and that the `int` value 0 casts to the `bool` value `false`.
This makes for a nice way to test if a pointer variable or parameter is currently pointing somewhere or not by directly testing for `nullptr` - but without using `ptrVar == nullptr`: (assuming that `ptrVar` is of some pointer type)

```
if( ptrVar )
```

The `bool` expression in the if statement test: ptrVar != nullptr in a nice C++ way.

The full amp and speaker setup code:

```
class SpeakerSystem {
   public:
   void vibrateSpeakerCones( unsigned signal ) const {

      cout << "Playing: " << signal << "Hz sound..." <<
endl;
      cout << "Buzz, buzzy, buzzer, bzap!!!\n";
   }
};

class Amplifier {
public:
   void attachSpeakers( const SpeakerSystem& spkrs )
   {
      if( attachedSpeakers )
         cout << "already have speakers attached!\n";
      else
         attachedSpeakers = &spkrs;
   }
   void detachSpeakers()
   { attachedSpeakers = nullptr; }
     // should there be an "error" message if not attached?
   void playMusic( ) const {
      if( attachedSpeakers )
         attachedSpeakers -> vibrateSpeakerCones( 440 );
      else
         cout << "No speakers attached\n";
   }
private:
   SpeakerSystem* attachedSpeakers = nullptr;
};
```

In the code above, you will find a problem compiling. What is it? Note what the method `attachSpeakers` is trying to do with the address of `spkrs`. *Why* is this a problem?

What do you think is the right way to fix it? There is more than one way and how you want to fix it depends on what you think the relationship will be between the amplifier and the speaker system. Know why you choose to pick one solution or the other is at least as important as making a change that gets the code working.

Create some `SpeakerSystem` and `Amplifier` variables.
Test by playing music on various speaker system configurations connected at various times to different speaker systems and trying to play some amps without speakers and by attaching another set of speakers to an already complete system (complete here means has speakers already attached - we're skipping the part about audio input devised.)
How do you cause a speaker system to become attached to an amplifier?

In a one-way association, the associated object (`SpeakerSystem` here) does not need to know anything about or communicate with the object it is associated with (`Amplifier` here).
If either of these situations is needed, then two-way association would be required (one-way would be the wrong design decision).
Are there any issues that might need to be considered that a one-way association cannot deal with in this problem?

In the debugger, when you get to a method call, do an F11 (Step Into) to see where execution goes.
Keep using F11 to see execution transfer to the `SpeakerSystem` method code.
Especially note the "`this`" pointer.

Task 30: Association between two objects of the same type (two-way association)

There should be a lot more checking before moving in with someone than is done here!

```
class Person {
public:
    Person( const string& name ) : name(name) {}
    void movesInWith( Person& newRoomate ) {
        roomie = &newRoomate;         // now I have a new
roomie
        newRoomate.roomie = this;     // and now they do too
    }
    string getName() const { return name; }
    // Don't need to use getName() below, just there for
you to use in debugging.
    string getRoomiesName() const { return roomie-
>getName(); }
private:
    Person* roomie;
```

```
    string name;
};

// write code to model two people in this world
Person joeBob("Joe Bob"), billyJane("Billy Jane");
// now model these two becoming roommates
joeBob.movesInWith( billyJane );
// did this work out?
cout << joeBob.getName() << " lives with " <<
joeBob.getRoomiesName() << endl;
cout << billyJane.getName() << " lives with " <<
billyJane.getRoomiesName() << endl;
```

Notice carefully that the person allowing the moving in with activity is responsible for setting the roommate field of their new roommate to themself - using their own "this" pointer to make the new roomie know who they moved in with.

The new roommate does not make any connections.

In:

`joeBob.movesInWith( billyJane );`

`joeBob` makes *both* "roommate" connections:

1) who his new roommate is

and

2) who his new roommate's (`billyJane`'s) roommate is (`joeBob`).

The programmer modeling the roommate relationships does not invoke any methods in `billyJane`.

In the debugger, when you get to the method calls above, do an F11 (Step Into) to see where execution goes.

Keep using F11 to see execution transfer.

Especially note the "this" pointer.