# Lecture 6

*Pointers*

"Absolute C++"
Section 10.1

## Pointers

- What is a pointer?
  - A pointer is a physical memory address which "points" at (presumably) an instance of a data type (either built-in or user defined)
  - A pointer variable "evaluates" to this address and is a way to pass a reference to the data type around without passing the data type itself.
  - A pointer variable to a given data type is declared by declaring a variable of that data type, except you precede the variable name with an asterisk

```
int *iPtr;  // Declares a pointer to int
```

- At this point, `iPtr` is a pointer to an `int` data type.
  - But it hasn't been initialized, so it doesn't point at anything
- You can do one of two things with it
  - Dynamically allocate space for a new `int` and store the result in `iPtr`
  - Assign an existing pointer value to it

## Dynamic Allocation

- We just showed how you declare a pointer variable, here's how you allocate space to it dynamically...

```
int *iPtr;
iPtr = new int;   // could also use new int();
```

- At this point iPtr contains one of the following:
  - A pointer to the newly allocated data type (in this case, an **`int`**)
  - NULL (if the pointer could not be allocated due to insufficient memory)
- You should always check for NULL before using a dynamically allocated pointer.  (there is another way to check, but that's later…)

```
int *iPtr = new int;     // Yes, this is legal
if (iPtr == NULL)
{
  // Report memory error here...
```

## Dynamic Allocation (cont)

- All dynamically allocated pointers stay "valid" until:
  - Your program terminates
  - You dispose of them
- How do you dispose of a dynamically allocated pointer?

```
int main()
{
  int *iPtr = new int;
  if (iPtr == NULL)
  {
    cout << "Could not allocate pointer, bye! ";
    return -1;
  }
  // Rest of program here
  delete iPtr;  // This is how you dispose of a pointer
  return 0;
}
```

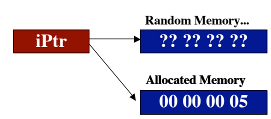
## Pointers: How To Access Content

- Access the contents of a pointer variable (the data it points to) by preceding the pointer variable with an asterisk.

```
int main()
{
  int* iPtr = new int;
  if (iPtr == NULL)
  {
    cout << "Could not allocate pointer, bye! ";
    return -1;
  }
  *iPtr = 5;  // Will actually write data into memory
  cout << "iPtr is " << iPtr << " and *iPtr is "
       << *iPtr << endl;
  delete iPtr;  // This is how you dispose of a pointer
  return 0;
}
```

## Pointers: How To Access Content

```
int main()
{
  int *iPtr;
  iPtr = new int;
  *iPtr = 5;
  cout << "iPtr is " << iPtr
       << " and *iPtr is "
       << *iPtr << endl;
  delete iPtr;
  return 0;
}
```




- First, the variable is declared.  At this point it points off into space (usually address 0)
- Second, space is allocated.  What is being pointed at is still undefined
- Third, a value is assigned
- Fourth, the value is retrieved and then the pointer is deleted.  The content cannot be trusted!

## Pointers: Allocating User Defined Types

- Everything we've just seen applies to classes too.
- Remember our `Course` class from previous lectures?

```
class Course
{
public:   // These can be seen outside the class
  // Define member functions
  string getCourseName();
  string getInstructor();
  int getStudentCount();
  void setCourseName(string theName);
  void setInstructor(string theInstructor);
  void setStudentCount(int count);

private:  // These can be seen inside the class only
. . .
};
```

## Pointers: Allocating User Defined Types

- We can define a pointer to it the same way we do for a built in type...

```
int main()
{
  Course *aCourse;
  aCourse = new Course;
  if (aCourse == NULL)  // Make sure we got the memory
  {
   cout << "Could not allocate memory for Course" << endl;
   return -1;
  }
   // Rest of program here…
  delete aCourse;
  return 0;
}
```

- But how do we access the member functions and variables?

## Pointers: Accessing Members via Pointers

- One way is to use the asterisk to *dereference* the pointer and then the period to get at the field:

```
Course *aCourse = new Course;
    (*aCourse).setStudentCount(45);
```

- Another way is to do both steps all at once with the `->` operator

```
Course *aCourse = new Course;
aCourse->setStudentCount(45);
```

- Let's take a look at this in action...

## Demonstration #1

Pointers to Classes

## Pointer Chaos

- What do you suppose the difference is between the following?

```
int *a,*b;
a = new int;
b = new int;
*a = 5;
*b = *a;
delete a;
cout << "b is " << *b << endl;
```

- and...

```
int *a,*b;
a = new int;
b = new int;
*a = 5;
b = a;
delete a;
cout << "b is " << *b << endl;
```

## Pointer Chaos (cont)

- Let's examine the second block more closely...

```
int *a,*b;
a = new int;
b = new int;
*a = 5;
b = a;
delete a;
cout << "b is " << *b << endl;
```

- Two things go wrong here towards the end of our code
  - We assigned the pointer `a` to the variable `b` and then deleted `a`.
    - This means that the actual pointer (memory address) stored in `a` was stored in `b`.
    - When we deleted `a`, `b` was left "dangling"
  - We changed the value of `b` without deleting the pointer it previously held
    - We lost any reference to that pointer, but it is still allocated!

# Demonstration #2

Pointer Chaos!

## Pointers to Existing Variables

- On top of being able to dynamically allocate and delete pointers to memory, we can also get a pointer to an existing variable.
- This is done with the `&` operator.

```
int main()
{
  int k, *iPtr;
  k = 5;
  iPtr = &k;

  cout << "k is " << k << " and *iPtr is " << *iPtr
       << endl;

  return 0;
}
```

- Let's take a look at this with our Course example:

# Demonstration #3

Using the & Operator

## Pointers to Existing Variables (cont)

- There are dangers...

```
int main()
{
  int *iPtr;
  if (true)
  {
    int p = 5;
    iPtr = &p;
  }
  cout << "*iPtr is " << *iPtr << endl;
}
```

- What happens here?
  - iPtr is set to point at the address of p.
  - At the end of the if statement, p goes out of scope.
  - iPtr is left pointing at unallocated (stack) memory.

## A Little About Stack Frames

- Whenever a new "scope" is encountered, C++ will allocate any local variables in that scope on the stack.
- Whenever a function is called a new "stack frame" is allocated on the stack which contains:
  - Space for all local variables in the function
  - Information on which function to return to when done
- Whenever a function is finished (`return` keyword encountered):
  - That function's stack frame is "removed"
- Consider the following function:

```
Course *MakeCourse(string name,string instructor,int size)
{
  Course aCourse;
  aCourse.setCourseName(name);
  aCourse.setInstructor(instructor);
  aCourse.setStudentCount(size);
  return( &aCourse );
}
```

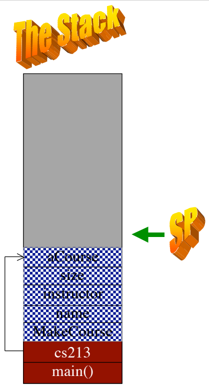
## Stack Frames (cont)

- Now consider that function being called like this:

```
int main()
{
  Course *cs213;
  cs213 = MakeCourse("COM S 213","DiNapoli",45);
  cout << "cs213->name = " << cs213->getCourseName() << endl;
  cout << "cs213->instructor = " << cs213->getInstructor()
       << endl;
  cout << "cs213->studentCount = << cs213->getStudentCount()
       << endl;
}
```

- What happens here?

## Stack Frames (cont)




```
int main()
{
  Course *cs213;
  cs213 = MakeNewCourse("COM S 213",
                        "DiNapoli", 45);
```

```
Course *MakeNewCourse(string name,
                      string instructor,
                      int size)
{
  Course aCourse;
  . . .
  return &aCourse;
}
```

```
// back in main()
cout << "cs213->name is " <<
     cs213->getCourseName() << endl;
```

BOOM!

# Lecture 6

*Final Thoughts*