Welcome to C++

Programming in C++

Slides from

Ron DiNapoli, Instructor

at Cornell University

Important Poll

- In order to get a feel for the experience of the class…
- *How many of you know Java?*
- *How many of you know C?*
- *How many of you know C++?*

This is not a complete C++ course

- An incomplete introduction to C++
- Focus on the basic difficulties with C++
  - Pointers, Pointers, Pointers
- Neglect all stuff, which is nice for OO-people, but may lead to inefficient code:
  - Abstract classes
  - Virtual methods, …

For More Information

- CS 213 Official Web Site:

http://salsa.cit.cornell.edu/cs213-sp01/index.html

- There are other useful links on the Web:
  – Use google :
    - Slides for C++ Courses
    - C++ Courses

A Simple C++ Program

```
#include <iostream>

void main()

{
  cout << "Hello World!" << endl;
}
```

- #include <iostreams> -- needed to access I/O streams (console)
- void main() -- main function-Entry point into your program
- {,} -- Scope delimiters
- cout -- the standard output identifier (console)
- << -- Special operator which takes contents to the right and sends them to the left
- endl -- special identifier which sends a newline

Some Simple C++ Type Declarations

```
int j;

float interestRate;

char aLetter;

string userName;
```

- int -- integer type: range is implementation dependent
  - usually 32-bits -- +/- 2,147,483,648
  - 16-bits on older systems -- +/- 32,768
- float -- floating point number
- char -- a single character
- string -- more than an array of characters (a class)

How to Assign Values

```
main()
{
    int j = 0;
    int k = 1;
    float pi;

    pi = 3.14159;
}
```

- Assignment at declaration time
  - insert an equals sign followed by an initial value
- Assignment of previously declared variable
  - start with the variable name, follow with equals sign, end with value to be assigned.

Arithmetic Expressions

```
main()
{
  int j = 0,k = 1,m = 2,n,p,q,r;
  float f;

  n = j + k;       // Add j and k, place in n
  p = n * m;       // Multiply n and m, store in p
  q = p / 4.0;     // Divide p by 4, place in q
  r = q - 1;       // Subtract 1 from q, place in r
}
```

- Can be used to calculate a value to be assigned
- What is wrong with the division expression?
- When assigning values to variables, value is always coerced to the type of variable it is getting assigned to.

Arithmetic Expressions (shortcuts)

```
main()
{
  int j = 0,k = 5;

  j++;              // really like j = j + 1;
  k -= 5;           // really like k = k - 5;
}
```

- When incrementing an integer variable by “1”, just append a “++” to the variable name.
- When decrementing by “1”, just append a “--” to variable name.
- When performing any other operation on a variable and stuffing the value back into the same variable, use a shortcut (like +=, -=, *=)

Arithmetic Expressions (prefix vs. postfix)

```
main()
{
  int j = 0,k = 0,q,r;

  q = j++;      // Postfix operation
  r = ++k;      // Prefix operation
}
```

- When “++” appears after variable it’s a “postfix operator”
  - the variable isn’t incremented until all other evaluations (and assignments) have taken place
- When “++” appears *before* variable it’s a “prefix operator”
  - the variable is incremented *before* any other evaluations take place.
- *What will the values of q & r be in the example above?*

Control Structures -- if/else statements

```
if (expression)
    statement1
else
    statement2
```

- *expression* is any expression that can be evaluated as an integer
  - “non zero value” is taken as “true”, “zero value” is taken as “false”
- *statement1* is a statement or group of statements executed if *expression* evaluates to a non-zero value
- *statement2* is a statement or group of statements executed if *expression* evaluates to a zero value
  - *statement2* is needed only if the the optional **`else`** keyword is present

Control Structures -- if/else statements

```
if (x = 0)
  cout << "It's zero" << endl;
else
  cout << "No, it's not zero!" << endl;
```

- WARNING!!!!!
  - While the "if" statement above may look perfectly fine
    it contains a very common flaw.
  - The assignment operator (=) is not used to test for equality.
  - "x=0" is an expression which evaluates to "0" along with having
    the side effect of storing the value 0 in the variable "x".
  - As an expression which evaluates to "0"= "false" it will always cause the "else" branch to be executed.

Control Structures -- if/else statements

```
if (x == 0)
  cout << “It’s zero” << endl;
else
  cout << “No, it’s not zero!” << endl;
```

- This is the correct way, use the equality operator (==)
- What are some of the other comparison operators?
  - (a > b), true if “a” is greater than “b”
  - (a < b), true if “a” is less than “b”
  - (a >= b), true if “a” is greater than or equal to “b”
  - (a <= b), true if “a” is less than or equal to “b”
  - (a != b), true if “a” is not equal to “b”

Control Structures -- compound expressions

```
if ((x == 0) || (y > 1))
{
  cout << "x is zero  OR" << endl;
  cout << "y is greater than 1" << endl;
}
```

- An expression with the logical "or" (||) operator…
  - –Evaluates to "true" if an expression on either side evaluates to "true"
- An expression with the logical "and" (&&) operator…
  - –Evaluates to "true" iff expressions on **both** sides evaluate to "true"
- Note the use of curly braces (`{,}`) above
  - –Used to group multiple statements to be executed if the "if" statement evaluates to "true"

Control Structures -- big if/else statements

```
int x;
cin >> x;                     // Hmmm, this is new!

if (x == 0)
  cout << “x is zero” << endl;
else if (x == 1)
  cout << “x is one” << endl;
else if (x == 2)
  cout << “x is two” << endl;
else
  cout << “x is not 0,1 or 2” << endl;
```

- Note the use of **`cin`** for input

Control Structures -- switch statement

A better way:

```
int x;
cin >> x;        // Hmmm, this is new!

switch(x)
{
  case 0:
     cout << “x is zero” << endl;
     break;
  case 1:
     cout << “x is one” << endl;
     break;
  case 2:
     cout << “x is two” << endl;
     break;
  default:
     cout << “x is not 0,1 or 2” << endl;
}
```

Don’t forget the breaks

Control Structures -- loops

```
while (expression)
      statement(s)
```

- A **`while`** loop will continue executing as long as ***`expression`*** evaluates to a non-zero (true) value.
- How do you print something 100 times using a while loop?

```
int x = 0;
while (x < 100)
{
  cout << "I have to do may homeworks" << endl;
  x++;
}
```

Control Structures -- loops

```
int x;
while (true)
{
  cin >> x;
  if (x == 0)
    break;
  cout << "You entered the number " << x;
}
```

- A **`while`** loop can be used to loop forever by having it test for an expression which will always evaluate to a non-zero value (true)
- A **`break`** statement can be used to break out of such a loop when the time comes
- Some say, this is bad programming style, but it is frequently used.

Control Structures -- loops

Say we need to loop 10 times:

```
for (int x=0; x<10; x++)   //local loop variable x
{
  cout << “Ronaldo Gool” << endl;
}
```

- Initialize -- `x = 0`
- Test -- `x < 10`
- Increment -- `x++`

CS 213 -- Lecture #2

Functions in C++

A function declaration:

```
return-type name ([arg-type name
                      {,arg-type name }])
```

- An optional return type is specified which tells the compiler the data type the function should return. If omitted, **`int`** is assumed per default.
- The name of the function. This name should be unique from all other function names.
- A comma separated list of type declarations specifying the parameters of the function. There may any number of parameters including 0.

Functions in C++

Let’s take a look at an example declaration:

```
long factorial(int n)
```

Declaration above has the following meaning:

- The return type is **`long`**. That means function **`factorial`** will return a long integer to the caller.
- The name of the function is **`factorial`**. When we need to call this function we will use this name.
- The function takes one parameter which is an integer variable named **`n`**.

Functions in C++

How might our factorial function be implemented?

```
long factorial(int n)
{
  long result = 1;            //intermediate result
  for (int k=n; k > 1; k--)
  {
    result *= k;
  }
  return result;
}
```

- Note the use of `long` for a local variable declaration.
- Note the use of a *decrement* in the for loop increment field.
- Note the use of `return` to return the value to the caller.

Functions in C++

How might we call our function from a main() function?

```
#include <iostream>
long factorial(int);
int main()
{
  int x;
  cout << “Please enter a number> “ << endl;
  cin >> x;
  cout << x << “! is “ << factorial(x) << endl;
}
```

- Note forward declaration
  - –Needed only if factorial() appears below main() in the file
  - –Note that parameters do not need to be specified but return type must!
- Function call--an expression which evaluates to its return value.
  - –Could also be used within an assignment

Argument Passing

- There are 2 ways to pass arguments to functions in C++:
  - Pass by VALUE
  - Pass by REFERENCE
- Pass by VALUE
  - The value of a variable is passed along to the function
  - If the function modifies that value, the modifications stay within the scope of that function.
- Pass by REFERENCE
  - A reference to the variable is passed along to the function
  - If the function modifies that value, the modifications appear also within the scope of the calling function.

Two Function Declarations

Here is a function declared as "pass by value"

```
long squareIt(long x)   // pass by value
{
  x *= x;    // remember, this is like x = x * x;
  return x;
}
```

- Now here is the same function declared as "pass by reference"

```
long squareIt(long &x)  // pass by reference
{
  x *= x;    // remember, this is like x = x * x;
  return x;
}
```

- What's the difference?

Two Function Declarations

```
#include <iostreams>
void main()
{
  long y;
  cout << "Enter a value to be squared> ";
  cin >> y;
  long result = squareIt(y);
  cout << y << " squared is " << result << endl;
}
```

- Suppose the user enters the number 7 as input
- When squareIt() is declared as pass by value, the output is:
  - 7 squared is 49
- When squareIt() is declared as pass by reference, the output is:
  - 49 squared is 49

Why use Parameter Passing By Reference?

- Because you *really* want changes made to a parameter to persist in the scope of the calling function.
  - The function call you are implementing needs to initialize a given parameter for the calling function.
  - You need to return more than one value to the calling function.

- Because you are passing a large structure
  - A large structure takes up stack space
  - Passing by reference passes merely a reference (pointer) to the structure, not the structure itself.

- Let's look at these two reasons individually...

Why Use Pass by Reference?

Because you want to return two values

```
void getTimeAndTemp(string &time, string &temp)
{
  time = queryAtomicClock();        // made up func.
  temp = queryLocalTemperature();  // made up func.
}
```

- All the caller would need to do now is provide the string variables

```
void main()
{
  string theTime, theTemp;
  getTimeAndTemp(theTime,theTemp);
  cout << "The time is: " << theTime << endl;
  cout << "The temperature is: " << theTemp << endl;
}
```

Why Use Pass by Reference?

Because you are passing a large structure:

```
void initDataType(BIGDataType &arg1)
{
  arg1.field1 = 0;
  arg1.field2 = 1;
  // etc., etc., assume BIGDataType has
  // lots of fields
}
```

- **`initDataType`** is an arbitrary function used to initialize a variable of type **`BIGDataType`**.
- Assume **`BIGDataType`** is a large class or structure
- With Pass by Reference, only a reference is passed to this function (instead of throwing the whole chunk on the stack)

Why Use Pass by Reference?

You can specify that a parameter cannot be modified:

```
bool isBusy(const BIGDataType &arg1)
{
  if (arg1.busyField = 0)
    return true;
  return false;
}
```

- By adding the **`const`** keyword in front of the argument declaration, you tell the compiler that this parameter must not be changed by the function.
- Any attempts to change it will generate a compile time error.

Scope

***OK**, we've used the "s" word a few times already today…*
What does it mean?

- Scope can be defined as a range of lines in a program in which any variables that are defined remain valid.
- Scope delimiters are the curly braces **{** and **}**
- Scope delimiters are usually encountered:
  - At the beginning and end of a function definition
  - In switch statements
  - In loops and if/else statements
  - In class definitions (coming next lecture)
  - All by themselves in the middle of nowhere

Scope

However, scope Delimiters may also appear by themselves...

```
void main()
{
  int x = 0,y = 1;
  {
    int x = 1, k = 5;
    cout << "x is " << x << ", y is " << y << endl;
  }
  cout << "x is " << x << " and k is " << k << endl;
}
```

- When you have multiple scopes in the same function you may access variables in any of the "parent" scopes.
- You may also declare a variable with the same name as one in a parent scope. The local declaration takes precedence.

Scope (cont.)

```
void main()
{
  int x = 0,y = 1;
  {
    int x = 1, k = 5;
    cout << "x is " << x << ", y is " << y << endl;
  }
  cout << "x is " << x << " and k is " << k << endl;
}
```

- What is wrong here?
- You may only access variables that are declared in the current scope or in an "outer or parent scope".
- Outside of main you may have global variables!!!

Function Declarations vs. Definitions

We've been somewhat lax about this...

```
long squareIt(long);    // Declaration
      .
      .
      .
long squareIt(long x)   // Definition
{
  return( x * x );
}
```

- Before a function may be called by any other function it must be either defined or declared.
- When a function is declared separately from its definition, this is called a forward declaration.
- Forward declarations need only to specify return type and parameter type. Parameter names are irrelevant.

Function Declarations and Header Files

- What happens when programs start getting really big?
  - We want to separate all functions we implement into logical groups. These groupings are usually stored in their own files.
  - How, then, do we access a function from one file when we are working in another file?

- We move the function declarations into header files

- Then all we need to do is include the appropriate header file in whatever source file needs it.

- Function definitions go into a source file with a **`.cpp`** suffix, function declarations go into a source file with a **`.h`** suffix

Function Declarations and Header Files

```
// mymath.h -- header file for math functions
long squareIt(long);
```

```
// mymath.cpp -- implementation of math functions
long squareIt(long x)
{
  return x * x;
}
```

```
// main.cpp
#include “mymath.h”
void main()
{
  cout >> “5 squared is “ >> squareIt(5) >> endl;
}
```

CS 213 -- Lecture #3

CLASSES

Classes

- What is a class?
  - "A class is a user-defined type"
  - Are all user-defined types classes?
    - No.
    - C++ supports the C notion of a "struct"
    - A struct allows programmers to define their own data type structures
      - Similar to RECORDs in Pascal
  - Are all classes user-defined types?
    - Yes
    - There are no "built in classes" in C++
    - There are provided standard class libaries
      - iostream
      - string
  - OK, that's great. But, what is a class?

Classes (cont)

- A class is a traditional data structure with a set of functions.
- Let's start with a simple C style structure definition

```
typedef struct{
  string name;
  string instructor;
  int numStudents;
} Course;
```

- Once defined I could use this "user defined" data type anywhere :

```
int main() {
  Course SA;     //SA new instance of Course
  SA.name = "System Architecture";
  SA.instructor = "Lieflaender";
  SA.numStudents = 300;
}
```

Classes (cont)

- Now, where do these functions fit in?
  - The functions (*member functions*) are tied to the data structure
  - Any “field” of the data structure may be accessed by any member function as if it were in a global scope.
  - Let’s take a look at this before we go any further...

```
class Course {
public:
  // Define member functions
  int getStudentCount() {  return numStudents; }

  // Define member variables
  string name;
  string instructor;
  int numStudents;
};
```

Classes: Public vs. Private

- Why bother with simple functions like getStudentCount() ?
  - It's a bad idea to directly access member variables
    - Circumvent error checking, easy to screw up data.
- Can't I just use the member variables directly anyway?

```
class Course {
public:   // These can be seen outside the class
  // Define member functions
  int getStudentCount() {  return numStudents; }

private:  // These can be seen inside the class only
  // Define member variables
  string name;
  string instructor;
  int numStudents;
};
```

Classes: Public vs. Private (cont)

- OK, so how do I access private data outside of the class?
  - You don’t, that’s the key idea of the *hiding principle*!!!
  - You can use get/set functions (public) to return the values for you

```
class Course {
public:   // These can be seen outside the class
  // Define member functions
  string getCourseName() { return name; }
  string getInstructor() { return instructor; }
  int getStudentCount() {  return numStudents; }
  void setCourseName(string theName)
  {  name = theName; }
  void setInstructor(string theInstructor)
  {  instructor = theInstructor; }
  void setStudentCount(int count)
  {  numStudents = count; }
private:  // These can be seen inside the class only
  ...
```

Classes: Lots of Member Functions

- Doesn't the class get unruly with all of those member functions?
  - Not really. The class definition only needs to have function declarations, not function definitions.

```
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 seed inside the class only
  ...
```

Classes: Lots of Member Functions

- Alright, but where do the member functions get defined?
  - Anywhere you want them to be defined :-)
  - No, seriously, with the help of some added notation they can be defined just about anywhere...

```
string Course::getCourseName()
{
return name;
}

int Course::getStudentCount()
{
return numStudents;
}
```

- Note the use of **`Course::`** to specify the class in question
- Note how I'm using member variables as if they were some sort of global variable

Classes: More on Public vs. Private

- The public and private labels can appear as many times as you want them to in a class definition.

```
class Course
{
public:   // These can be seen outside the class
  // Getter functions
  string getCourseName();
  string getInstructor();
  int getStudentCount();
public:
  // Setter functions
  void setCourseName(string theName);
  void setInstructor(string theInstructor);
  void setStudentCount(int count);
private:  // These can be seed inside the class only
  // Member variables
  ...
```

Classes: More on Public vs. Private

- Member functions can be private as well.

```
class Course
{
public:   // These can be seen outside the class
  // Getter and Setter 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
  // private member functions
  bool validateStudentCount(int count);
  ...
```

Classes: More on Public vs. Private

- You can still have public member variables
- If no public or private label is specified, private is assumed

```
class Course {
  bool validateStudentCount(int count);  // implicit private

public:
  bool isFull;   // publicly accessible member variable

  // Getter and Setter functions
  string getCourseName();
  string getInstructor();
  int getStudentCount();
  void setCourseName(string theName);
  void setInstructor(string theInstructor);
  void setStudentCount(int count);
...
```

Where Should We Define Member Functions?

- *How do you know when to define a member function in the class definition vs defining it outside of the class definition?*
- A good rule of thumb is:
  - If the definition is simple (one line of code) you should define it in the class definition.
  - Getter/Setter functions are prime examples of simple functions.
  - Otherwise, define outside of class definition, in a separate file.

What Files Should These Definitions Go In?

```
// Course.h -- Header file for Course class
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
  string name,instructor;
  int count;
};
```

What Files Should These Definitions Go In?

```
// Course.cpp -- Definition file for Course class
#include "Course.h"                      //necessary !!!!
string Course::getCourseName()
{
  return name;
}
String Course::getInstructor()
{
  return instructor;
}
String Course::getStudentCount()
{
  return count;
}
```

CS 213 -- Lecture #4

POINTERS

The most awful idea about C++,

but systems' programmer need them

Pointers

- *What is a pointer?*
  - Pointer is a *physical* memory address which "points" at an instance of a data type (either built-in or user defined)
  - 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.
  - 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`**, store result in **`iPtr`**
  - Assign an existing pointer value to it

Pointers: Dynamic Allocation

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

```
int *iPtr;          // already 4 bytes for the pointer
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 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 also legal
if (iPtr == NULL)
{
  // Report memory error here...
```

Pointers: 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;
}
```

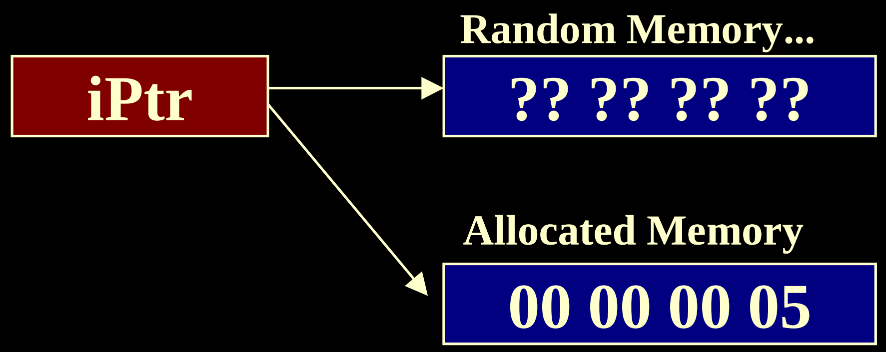
Pointers: How To Access Content

- 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, 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 (at last)
- Fourth, value is retrieved, 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 example

```
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 seed 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;      // Declaration of a pointer
  aCourse = new Course; // memory for object of class
  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);
```

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;                                  // the mess begins
delete a;                               // the mess is complete
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**, now **b** has been left “dangling”
  - We changed value of **b** without deleting the pointer it held before
    - We lost any reference to that pointer, but it is still allocated!

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;    // only 2 declarations
  k = 5;           // at least k has now a value
  iPtr = &k;       // and now also iPtr has its value

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

  return 0;
}
```

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

Pointers to Existing Variables (cont)

- However, even in this case there are some 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 its 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” (i.e. no longer valid)
- 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)

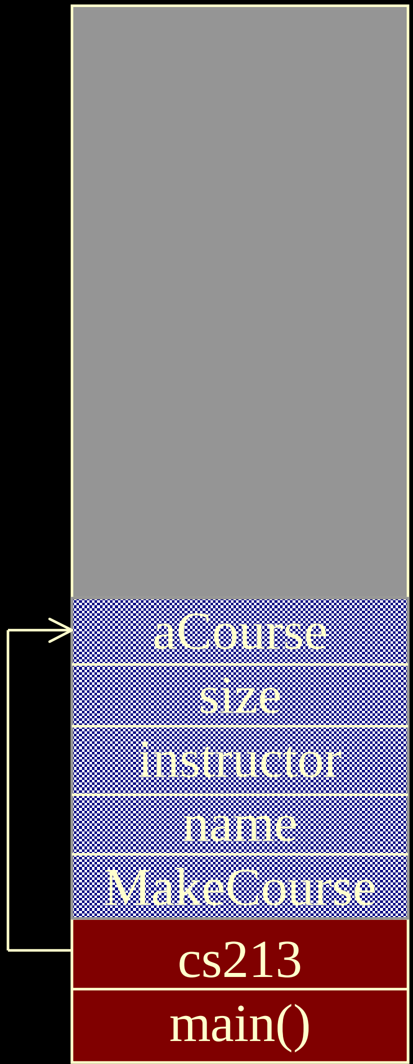
- Now consider that function being called like this:

```
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)

The Stack




```
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!