UMassAmherst

CS197c: Programming in C++

Lecture 5

Marc Cartright

http://ciir.cs.umass.edu/~irmarc/cs197c/index.html

Syllabus

- ~~Lecture 1 : C/C++ basics, tools, Makefiles, C data types, ADTs~~
- ~~Lecture 2 : C libraries~~
- ~~Lecture 3 : Classes in C++, C++ I/O~~
- ~~Lecture 4 : Memory & Pointers~~
- Lecture 5 : More Pointers
- Lecture 6 : Templates and the STL
- Lecture 7 : Reflection, Exceptions, C++11
- Lecture 8 : Adv. Topics: Boost and OpenGL

Schedule for Today

- Form groups of 2-3
- ~20 minutes working on the problems
- Review Problems
- Quick lecture on Pointers & Classes
- Q &A

QUIZ TIME!!!

1

- `int **p = &&n;`

1

- `int **p = &(&n);`

This makes no sense – how can you take the address of just an address?

2

```
double *f();                double (*f)();
```

2

double *f();

double (*f)();

This function is called ‘f’, takes no arguments, and returns a pointer to a double.

Example:

```
#include <cmath>
double *dblptr;

dblptr = f();
double cosine = cos((*dblptr));
```

2

```
double *f();                    double (*f)();
```

This is a pointer to a function (the pointer name is ‘f’), and it returns a double.

Example:

```
#include <cstdlib>
f = &drand48();          // our function pointer now looks at
                         // drand48 – PRNG for doubles.
double result = f();
```

3

```
float x = 3.14159;
float *p = &x;
short d = 44;
short *q = &d;
p = q;
```

3

```
float x = 3.14159;
float *p = &x;
short d = 44;
short *q = &d;
p = q;
```

Can’t assign a pointer of one type to another unless you’re casting up in the class hierarchy.

4

- What is the code you would use to allocate a 2-D primitive array of `int`s of size *m* by *n*?

  Hint: You will need to use double pointers -- pointers that point to pointers.

4

- What is the code you would use to allocate a 2-D primitive array of ints of size *m* by *n*?

  Hint: You will need to use double pointers -- pointers that point to pointers.

```
int **arr = new int*[m];
for (int i = 0; i < m; i++) {
  arr[i] = new int[n];
}
```

5

- Delete the 2-D array.

```
int **arr = new int*[m];
for (int i = 0; i < m; i++) {
  arr[i] = new int[n];
}
```

5

- Delete the 2-D array.

```
int **arr = new int*[m];
for (int i = 0; i < m; i++) {
  arr[i] = new int[n];
}
for (int i = 0; i < m; i++) {
  delete [] arr[i];
}
delete [] arr;
```

6

Random selection of student answers!

7

```
int m = 44;
int *p = &m;
int &r = m;
int n = (*p)++;
int *q = p - 1;
r = *(--p) + 1;
++*q;
```

(i) m (ii) n (iii) &m (iv) *p (v) r (vi) *q

7

```
int m = 44;
int *p = &m;
int &r = m;
int n = (*p)++;
int *q = p - 1;
r = *(--p) + 1;
++*q;
```

Ok, there's a lot going on here...
Let's go line by line!

(i) m (ii) n (iii) &m (iv) *p (v) r (vi) *q

7

```
int m = 44;
int *p = &m;
int &r = m;
int n = (*p)++;
int *q = p - 1;
r = *(--p) + 1;
++*q;
```

Var	Value
m	?
n	?
&m	?
*p	?
r	?
*q	?

7

```
int m = 44;
int *p = &m;
int &r = m;
int n = (*p)++;
int *q = p - 1;
r = *(--p) + 1;
++*q;
```

Var	Value
m	44
n	?
&m	0x3fffd00
*p	?
r	?
*q	?

m is allocated at address 0x3fffd00, and assigned value 44.

7

```
int m = 44;
int *p = &m;
int &r = m;
int n = (*p)++;
int *q = p - 1;
r = *(--p) + 1;
++*q;
```

Var	Value
m	44
n	?
&m	0x3fffd00
*p	44
r	?
*q	?

p is allocated at address 0x3fffd04, and assigned pointer value 0x3fffd00 (addr. of *m*).

7

```
int m = 44;
int *p = &m;
int &r = m;
int n = (*p)++;
int *q = p - 1;
r = *(--p) + 1;
++*q;
```

Var	Value
m	44
n	?
&m	0x3fffd00
*p	44
r	44
*q	?

r is allocated at address 0x3fffd08, and also assigned reference value 0x3fffd00.

7

```
int m = 44;
int *p = &m;
int &r = m;
int n = (*p)++;
int *q = p - 1;
r = *(--p) + 1;
++*q;
```

Var	Value
m	45
n	44
&m	0x3fffd00
*p	45
r	45
*q	?

n is allocated at address 0x3fffd0c. We first dereference *p*, retrieving value 44. Since we're using the postfix increment, the value returned is the value *before* incrementing, therefore *n* is assigned 44, and *then* *m* is incremented to 45.

7

```
int m = 44;
int *p = &m;
int &r = m;
int n = (*p)++;
int *q = p - 1;
r = *(--p) + 1;
++*q;
```

Var	Value
m	45
n	44
&m	0x3fffd00
*p	45
r	45
*q	*(0x3fffcfc)

q is allocated at address 0x3fffd10. We use pointer math here, and take the address stored at *p* and decrement it by one according to the base type, which is int. Therefore, we subtract 4 from the address stored at *p*, and *q* is assigned 0x3fffd00 - 4 = 0x3fffcfc.

7

```
int m = 44;
int *p = &m;
int &r = m;
int n = (*p)++;
int *q = p - 1;
r = *(--p) + 1;
++*q;
```

Var	Value
m	45
n	44
&m	0x3fffd00
*p	45
r	45
*q	*(0x3fffcfc)

We first decrement *p* (prefix decrement), then take the value at the address, which is (&*m* - 4), and add one to it. If *x* is the value at address 0x3fffcfc, then *r*, and therefore *m*, are assigned the value x+1.

7

```
int m = 44;
int *p = &m;
int &r = m;
int n = (*p)++;
int *q = p - 1;
r = *(--p) + 1;
++*q;
```

Var	Value
m	(*q)+1
n	44
&m	0x3fffd00
*p	(*q)+1
r	(*q)+1
*q	*(0x3fffcfc)

We first decrement *p* (prefix decrement), then take the value at address (&*m* - 4), and add one to it. *r*, and therefore *m*, are now set to that value.

7

```
int m = 44;
int *p = &m;
int &r = m;
int n = (*p)++;
int *q = p - 1;
r = *(--p) + 1;
++*q;
```

Var	Value
m	x+1
n	44
&m	0x3fffd00
*p	x+1
r	x+1
*q	*(0x3fffcfc)

(variable substitution to decouple)

7

```
int m = 44;
int *p = &m;
int &r = m;
int n = (*p)++;
int *q = p - 1;
r = *(--p) + 1;
++*q;
```

Var	Value
m	x+1
n	44
&m	0x3fffd00
*p	x+1
r	x+1
*q	x+1

Dereference *q* and increment the value.

We've seen

- Pointers

```
int n = 32;
int *p = &n;
```

- Classes

```
class Student {
public:
	int numClasses;
	…
```

So let's put them together

- Pointers and classes are extremely useful together:

```
class Student {
  public:
        Student();
        ~Student();
  private:
        vector<Course *> courses;       // vector of courses
};
```

Why bother?

- Better control
- More efficient (memory)
- More efficient (speed)

Passing Pointers

A function can return the address of an object that was created on the heap. In this example, the function's return type is pointer to Student.

```
Student *makeUpNewStudent() {
        Student *s = new Student();
        s->firstName = “Pablo”; return s;
}
```

Passing Pointers (cont'd)

- (continued)... The caller of the function can receive the address and store it in a pointer variable. As long as the pointer remains active, the Student object is accessible.

```
Student *makeUpNewStudent() {
        Student *s = new Student();
        s->firstName = "Pablo"; return s;
}
Student *sPtr = makeupNewStudent();
```

Semantics with Pointers

- A const-qualified pointer guarantees that the program has read-only access to the data referenced by the pointer:

  ```
  const int * ptr;
  ```

- Declaring a constant pointer guarantees only that the pointer itself cannot be modified: `int * const ptr;`

Pointers and Classes

- Pointers are effective when encapsulated in classes, because you can control the pointers' lifetimes.
- The constructor creates the array, and the destructor deletes it. Very little can go wrong,...

Pointers and Classes (cont'd)

- ...except when making a copy of a Student object. The default copy constructor used by C++ leads to problems. In the example, a course assigned to student X ends up in the list of courses for student Y. (demo)

Copying with classes

- Hmm...that didn't go according to plan
- How can we fix it?

Copying with classes

- Solution:

  Explicitly defining the copy constructor!

```
Student::Student(Student s) {
  //deep copy course items here
}
```

Next Lecture

Templates

&

The Standard Template Library (STL)

Now: Q&A on PA2