CS368 Lecture 4

Wednesday 24 September 2014

Reminders:

- P1 due 11 PM Friday
- HW1 assigned
- Readings

Last class:

- Vectors
- Pointers and Reference Variables
- Parameter Passing

Today:

- Pointers (cont'd)
    - Practice
    - Passing Pointers as Arguments
- Dynamic Allocation
- Pointers and Arrays
- Pointer caveats

Practice with Pointers and References

```
int x = 9;
int* ptrToX = &x;
int& refToX = x;
```

```
int a = 15;
int *p = &a;
```

What do we see if we print out

- `a`
- `*p`
- `&a`
- `p`

```
int b = 17;
p = &b;
*p = 20;
```

What do we see if we print out

- `b`
- `*p`
- `&b`
- `p`

Passing Pointers as Arguments

```
void foo(int* px) {
    *px = *px + 1;
    print("px", px);
}
```

In `main()`:

```
int a = 5;
foo(&a);
print("a", a);
```

The Stack and the Heap

Dynamic Memory Allocation

new:

Make room on the heap.

```
int* p = new int;
int* q = new int(); // parentheses optional
int* r = new int(37);
Book* b1 = new Book;
```

delete:

Free (dynamically allocated) memory space after use.

```
delete p;
cout << p << ", " << *p << endl;
```

Pointing to NULL

Why set things to NULL? Disallowing future access.

```
p = NULL;
cout << p << ", " << *p << endl;
```

Practice with Pointers (2): Structs

```
struct Patron {
     string name;
     int libraryID;
}

struct Book {
     int bookID;
     int numCheckouts;
     Patron borrower;
}

int main() {

     Book b1 = {234, 12, {“Tim”, 10}};

     Book* p2 = new Book;
     p2->bookID = 392;
     p2->numCheckouts = 0;
     p2->borrower.name = “Kate”;
     p2->borrower.libraryID = 12;

     Book b3 = *p2; // What gets copied here?
     b3.bookID = 300;

     Book* p4 = &b1;

     Book* p5 = p2;


     cout << p2->borrower.name << endl;
     cout << (*p2).borrower.name << endl; // Identical

     cout << b3.bookID << endl; // NOT same as p2->bookID!
     cout << p4->bookID << endl; // SAME as b1.bookID
     cout << p5->borrower.name << endl; // SAME as p2->...

     delete p2; // delete only the one created using new
     p2 = NULL; // reset the pointer to prevent misuse

     return 0;
}
```

Arrays are Pointers

```
int arr[4] = {2, 4, 6, 8};
int* p = new int[5];

for (int i=0; i<5; i++)
    p[i] = 2 * arr[i]; // treat p like an array

int* q = arr; // q points to the same array now

*q = 12; // another way of accessing arr[0]

*(q+1) = *(p+2); // setting arr[1] equal to p[2] (= 12)
                 // another way of traversing the array
```

Note: Pointer increment is based on the size of the type pointed to:

```
cout << q << ", " << q+1 << ", " << q+2 << endl;
```

Verify by printing:

```
cout << q[0] << ", " << *arr << endl;
cout << arr[1] << ", " << p[2] << endl;
```

Pointer Caveats

Checking for equality:

```
int* p = new int(5);
int* q = new int(14);
*p = *p + 9;
cout << p == q << endl;
cout << *p == *q << endl;
```

Dereferencing NULL, uninitialized, deleted pointers:

```
int* ptr; // Uninitialized
*ptr = 10;

int* ptr = &x;
ptr = NULL
*ptr = 10;

delete ptr;
*ptr = 15;
```

Memory leaks:

```
int* p = new int(30);
p = NULL;
…
p = new int(40);
```

Delete is only for dynamically allocated memory:

```
int x = 20;
int* p = &x;
delete p;
```

Delete a block of memory only once:

```
int* p = new int(20);
int* q = p;
delete p;
delete q; // will crash!
```

On Your Own

- Work through pointerBasics.cpp, work out the output and verify
- Modify, compile, and run the array sample code above
- Run and check pointer caveats on your own