CSE 143

Pointers, Arrays, and Dynamic Storage Allocation

[Chapter 4, pp. 148-157, 172-177]

Storage Allocation

- Storage (memory) is a linear array of cells (bytes)
- Objects of different types often require differing amounts of storage
- Built-in types: implementation dependent
  - PC (typical):

```
char: 1 byte (8 bits)
int: 4 bytes (2 bytes on older systems)
double: 8 bytes
```

- Programmer defined types: depends on size of data members
  - could be few bytes or thousands of bytes

Pointers: Review

- Every memory cell in a computer's memory has two characteristics:
  - A value (contents)
  - A unique address
- Every object in the program occupies a range of memory cells
- Suppose we have:

```
int x;
```

  - Value (contents) of x:

```
x
```

  - Address (location) of x:

```
&x
```


Pointer Variables

- By "address of an object" we mean the address of the first memory cell occupied by the object
- A pointer variable is one that contains the address of another data object as its value.
- Pointer variable declaration:

```
Type* name;
```

- Examples:

```
int* intPtr;
char* charPtr;
BigNat* bigNatPtr;
```


Pointers and Types

- Pointers to different types themselves have different types

```
double *dpt;
Cell * cp;
```

- In C/C++, **dpt** and **cp** have different types
  - even though under the hood they are both just memory addresses
- Types have to match in many contexts
  - e.g. parameter and argument types in a function call
  - pointers are no exceptions

Pointer Operations

- Operations valid on pointers of all types. We'll cover only a subset:

= (pointer assignment)

```
int* p = &someInt;
```

= (pointer assignment)

```
int* q = p;
```

* (dereference)

```
*p = *p + 1;
cout << *q;
```

More Pointer Operations

== (equality test)

```
if (ptr1 == ptr2) { . . . }
```

!= (test for inequality)

```
if (ptr1 != ptr2) { . . . }
```

-> (select a member of a pointed-to struct or object)

```
void foo (Complex* b) {
  b->re = -1;
}
```


Pointers and Arrays

In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously. Any operation that can be achieved by array subscripting can also be done with pointers.

Brian Kernighan and Dennis Ritchie
The C Programming Language

In C++, pointers and arrays are very closely related.

Bjarne Stroustrup
The C++ Programming Language

Arrays vs. Pointers

- An array name refers to the address of the first element of the array
- Array notation and pointer notation can be mixed

```
int a[10];      // int array
int *pa, *pb;   // int pointers
pa = &a[0];
*pa = 5;
pb = pa;
j = *pb;
```


Arrays ≠ Pointers!

```
int * ip;
int iarr[10];
// what happens now?
iarr[0] = 100;
ip[0] = 200;
ip = iarr;
iarr = ip;
ip = new int[20];
iarr = new int[20];
```


Arrays Parameters

- Array and pointer parameters are interchangeable
- These four functions are exactly the same

```
void seta0(char a[ ]) { a[0] = 17; }
void seta0(char a[ ]) {  *a  = 17; }
void seta0(char *  a) { a[0] = 17; }
void seta0(char *  a) {  *a  = 17; }
```

- Use whichever makes your code easier to read and understand

Variable-Size Data

- Problem: All of our data structures so far have a "maximum" size.
- This size is fixed at *compile time*.
- Sometimes this is ok, but most real applications need to grow and shrink the amount of memory consumed by a data structure *during execution*.
- Most languages provide some form of dynamic memory.
- C++ provides an interface to dynamic memory via two operators: **`new`** and **`delete`**.

Storage Classes

- *Static* - allocated at program startup time, exists throughout the execution of the entire program
- *Automatic* - implicitly allocated on function entry, deallocated on exit

```
void foo (char x) {
  int temp;
  . . .
  // x and temp are deallocated here
}
```

- *Dynamic* - explicitly allocated and deallocated by the programmer

Allocation & Deallocation

- Allocate dynamic memory with operator **new**:
  - The expression `new Type` returns a pointer to a newly created object of type `Type`:

```
int* p;
char* str;
p = new int;         // allocate a single int
str = new char[10];  // allocate an array of chars
```

- Deallocate memory with operator **`delete`**:
  - `delete Pointer` deallocates the object pointed to by `Pointer`

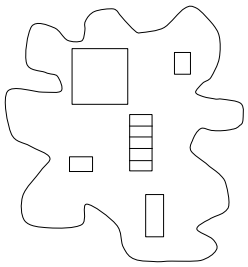
```
delete p;        // deallocate a simple item
delete [] str;   // deallocate an array
```


The Heap

- Objects created by **new** come from a region of memory set aside for dynamic objects (called the *heap*, or *free store*).
- The **new** operator obtains a chunk of memory from the heap; **`delete`** returns that memory to the heap.
- In C++ the programmer must manage the heap.
- Dynamic data is unnamed (anonymous) and can only be accessed through pointers.

Heap Memory

```
int *v, *w;
v = new int;
w = new int[5];
IntSet * ps;
ps = new IntSet;
delete v;
delete [] w;
delete ps;
```



Constructors, arrays, & **new**

- Whenever a class instance is created, a constructor is called

```
IntSet s;       // initialized by IntSet::IntSet( )
```

- This is true for all elements in an object array

```
IntSet a[3];    // IntSet::IntSet( ) executed 3x to
                //   initialize each of s[0],s[1],s[2]
```

- The same rules apply when objects are allocated dynamically.

```
IntSet *sp1 = new IntSet;    // constructor called once
IntSet *sp2 = new IntSet[3]; // constructor called for
                             //   each array element
```


The NULL pointer

- During program execution, a pointer variable can be in one of the following states:
  - Pointing to a data object
  - Unassigned (never initialized or pointing to deleted storage)
  - Contain the special value NULL
- The constant NULL is usually defined as 0 in <stddef.h>, and is used to mean a pointer that does not point to any object.
- NULL is compatible with all pointer types

Dynamic Memory Dangers

- Failure to return objects to heap
  - Run out of resources

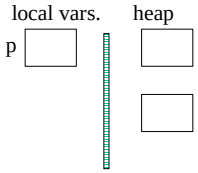
```
IntSet *ps;
for (int i = 0; i < 100000000; i++)
  ps = new IntSet;
```

- Pointers to non-allocated objects
  - Accessing garbage values
  - Overwriting valid data
  - Causes:
    - Non initialized or null pointers
    - Keeping references to deleted objects
    - Pointer arithmetic
- Misusing delete
  - Deleting deleted data
  - Confusing **`delete p`** and **`delete [ ] p`**

Uninitialized Pointer

- Example

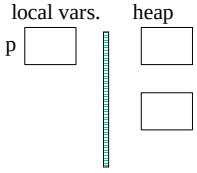
```
int* p;
*p = 45;
```



Garbage (memory leak)

- Example 1

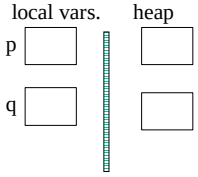
```
int* p;
p = new int;
*p = 45;
p = new int;
*p = 55;
```



Garbage (memory leak)

- Example 2

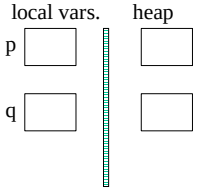
```
int *p, *q;
p = new int;
q = new int;
*p = 45;
*q = 55;
p = q;
```



Dangling Pointers

- Example 1

```
int *p = new int;
int *q;
*p = 45;
q = p;
delete p;
*q = 55;
```



Dangling Pointers

- Example 2

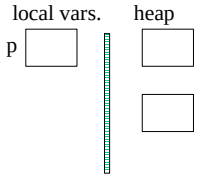
```
char* broken( ) {
  char buffer[80];
  cin >> buffer;
  return buffer;
}
charPtr = broken( );
  // charPtr is dangling
```

Destroyed when function exits

Abusing **delete**

- Example 1

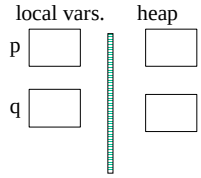
```
int *p
p = new int;
*p = 45;
...
delete p;
...
delete p;
```



Abusing **delete**

- Example 2

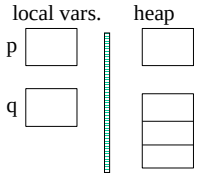
```
int *p, *q;
p = new int;
*p = 45;
q = p;
...
delete p;
delete q;
```



Abusing **delete**

- Example 3

```
int *p, *q;
p = new int;
q = new int[3];
...
delete [ ] p;
delete q;
```



Dynamic Data Guidelines

- Avoid creating garbage when invoking **new** or moving pointers.
- Operator **new** returns a pointer to an object. If the object's type is a class, then the constructor has been called. If not, the object is still uninitialized.
- Don't dereference an unassigned pointer.
- Don't dereference a NULL pointer.
- After **delete**, the pointer value is unassigned (don't assume NULL)