CSE 143

Dynamic Memory

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

What's wrong with the way things are?

- One problem: All of our data structures so far have a "maximum" size.
  - E.g. arrays declared with fixed size
- This size is fixed at *compile time*.
- Sometimes this is acceptable, sometimes not
  - Allocate too little: application may not run
  - Allocate too much: wasted memory (may run out)
- Many real applications need to grow and shrink the amount of memory consumed by an object at *run time*.

A "Shape" Problem

- All of our data structures so far are fixed in form and shape
  - Individual vars, structs, classes, or arrays of them, or simple nesting
- Many problems require more creative shapes
  - Family tree
  - Company database
  - Recursive data, complex links
- Needed variety
  - for modeling the data
  - for efficiency

Solution: "Dynamic" Memory

- 1. Allow some of the memory to be allocated *as needed*
- 2. Allow pieces of memory (variables) to be linked in arbitrarily complex ways
- Most languages provide some form of dynamic memory.
- C++ provides an interface to dynamic memory via two new operators: **new** and **delete**.
  - The dynamic memory is accessed through pointers.

Plan of Study

- First
  - Review pointers and reference parameters
- Next
  - Introduce C++ new and delete operators
  - Dangers!
  - Dynamic memory in classes
  - Pointers vs. arrays
  - Dynamic linked lists
- Finally...
  - Even more about dynamic memory in classes
  - Vector class revisited

Data and Memory

- Objects of different types use differing amounts of memory
- Built-in types: implementation dependent
  - PC (typical):
    - char: 1 byte (8 bits)
      - "wide" chars: 2 bytes (for international UNICODE)
    - int: typically 4 bytes
      - 2 bytes on older systems
      - up to 8 bytes on newest "64-bit" computers
    - double: 8 bytes on many systems
- Programmer defined types (such as classes)
  - depends on size of data members
  - could be few bytes or thousands of bytes

Ways of Using Memory

- *Static* - allocated at program startup time, exists throughout the execution of the entire program
  - Best-known example: global variables
- *Automatic* - implicitly allocated upon 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

Pointer Variables

- By "address of an object" we mean the address of the first memory cell used by the object
- A pointer variable is one that contains the address of another data object as its value.
- To declare a pointer variable or param:

```
Type* name;
```

- Example:

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


Review: Swap in C

- In CSE 142, you used pointers to write functions which modified their actual parameters:

```
void swap(int* p, int* q) {
   int temp;
   temp = *p;
   *p = *q;
   *q = temp;
}

// example call:
swap(&intOne, &intTwo);  // don't forget the &
```


Two Important Operators

- The address-of operator &:

```
int x = 45;
int* p = &x;
```

- The dereference operator *:

```
*p = 30;
p = 72;    // what's the problem here?
```

Note: *The & symbol used with reference parameters is the same keyboard character, but it means something quite different in that context*

Review: Swap in C++

- C++ lets us use reference parameters, leading to cleaner code:

```
void swap(int& a, int& b) {
   int temp = a;
   a = b;
   b = temp;
}

// example call:
swap(intOne, intTwo);  // note:  no &
```


Reference Types

- Main use: for parameters
- We can also declare variables of reference types:

```
Type& rname        //rname will hold an alias to something
                   //of type Type
```

- Example:

```
int x;
int& refx = x;  // a ref. variable must be initialized

x = 40;
cout << refx;   // what's the output?
refx = 20;
cout << x;     // what's the output?
```

- In 143 we will avoid stand-alone reference variables
  - but reference params are used as needed.

Pointers and Types

- Pointers to different types themselves are different types

```
double *dpt;
BankAccount * bp;
```

- C/C++ considers dpt and bp to have *different* types
  - even though under the hood they are both just memory addresses
- Types have to match in many contexts
  - e.g. actual param types matching formal param types
  - pointers are no exceptions

3/25/2001 J-13

C++ Is "Strongly Typed"

```
int i;  int * ip;
double x; double * xp;
...
x = i;        /* no problem */
i = x;        /* not recommended */

ip = 30;      /* No way */
ip = i;       /* Nope */
ip = &i;      /* just fine */
ip = &x;      /* forget it! */
xp = ip;      /* bad */
&i =  ip;     /* meaningless */
```

3/25/2001 J-14

The NULL pointer

- During program execution, a pointer variable can be in one of the following states:
  - Unassigned (uninitialized)
  - Pointing to a data object
  - Contain the special value NULL (can also use 0)
- The constant NULL is defined as 0 in <cstddef> (stddef.h), and is used to mean "a pointer that does not point to any object."
  - It does not mean "address 0 of the computer"
- NULL is compatible with all pointer types

3/25/2001 J-15

Pointers as Types

- Domain (possible values)
  - The set of all memory addresses along with the NULL pointer
- Some operations are valid on pointers of all types. We'll cover only a subset:

= (assignment)

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

* (dereference)

```
*p = 345;
```

== (equality test)

```
if (ptr1 == ptr2) { . . . }
  //Carefull!! What is being compared?
```

3/25/2001 J-16

More Pointer Operations

!= (test for inequality)

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

delete (deallocate)

```
delete ptr;  // more on this later
```

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

```
void foo (BankAccount* b) {
  b->printBalance();
}
// How would you write this if -> were not available?
```

3/25/2001 J-17

new: Allocating Memory

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

```
int *p, *p2;
p = new int;          // allocate a single int
*p = 2001;
p2 = new int[10];  // allocate an array of ints
p2[0] = -17;        // can use array notation with ptrs
```

- The memory allocated will be the right size for the type of object
  - The pointer locates the beginning of that memory area.

3/25/2001 J-18

new Could Fail!

*int * bigP = new int [1000000];*

- new returns NULL if the memory could not be allocated (or throws an exception in newer versions of C++)
- Advice: always test result
  - Assert is simple:

*int * bigP = new int [1000000];*
assert (bigP != NULL);

  - or make a test before using:

if (bigP != NULL) ... // go ahead and use the pointer
else ... // take some recovery action

3/25/2001 J-19

Deallocation

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

```
delete p;        // deallocating a simple object
delete [] str;   // deallocating an array of objects
```

  - The proper amount of memory is released
- Delete does ***not*** alter the bits in the pointer!
  - Useful habit:

```
delete p;   // p not changed
p = NULL;
```

- The memory MUST have been allocated via *new*
  - Woe if you try to delete local memory, etc.
  - Disaster if you use delete instead of delete[ ] or vice versa

3/25/2001 J-20

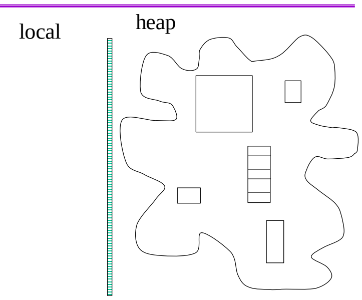
Where does the memory come from?

- Objects created by **new** come from a region of memory set aside for dynamic objects
- Sometimes called the *heap*, or *free store*
  - Textbook doesn't use those names
- 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 memory is unnamed and can only be accessed through pointers.

3/25/2001 J-21

Heap Memory

```
int *v, *w;
v = new int;
w = new int[5];
BA *pBA;
pBA = new BA;
delete v;
delete [] w;
delete pBA;
```



3/25/2001 J-22

Dynamic Memory: Review So Far

- *new* gets memory, *delete* gives it back
- In all cases: The ***new*** operator returns a pointer to an object.
  - Unless new fails -- then returns NULL (or throws an exception, which probably terminates the program)
- The memory is on the heap
  - unlike local variables, which are in the activation record (stack frame)

3/25/2001 J-23

Dynamic Memory Is Dangerous

- A major source of program bugs
  - Memory leaks: not giving back allocated memory
  - Dangling pointers: using a pointer to memory no longer allocated
    `may silently clobber data`
  - Using uninitialized pointers
    `may silently clobber data`
  - Security violations: giving client access to private data
- These are run-time errors
  - Compiler can't catch them
  - The program may appear to run correctly... sometimes

3/25/2001 J-24

A Quote from Bjarne Stroustrup

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows your whole leg off."

3/25/2001 J-25

Memory Leak Example

- Failure to return objects to heap ("memory leak")
  - Computer might run out of resources

```
BankAccount *pBA;
for (int i = 0; i < 100000000; i++)
  pBA = new BankAccount;
```

- “Garbage:” allocated memory for which there is no pointer
- It’s not always this obvious!

3/25/2001 J-26

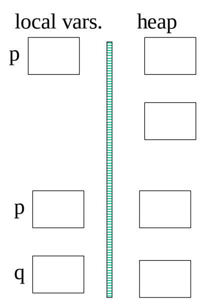
Garbage (Memory Leak)

- Example

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

- Example 2

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

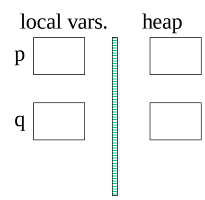


3/25/2001 J-27

Dangling Pointers

- Example 1

```
int *p = new int;
int *q;
*p = 45;
q = p;
delete p;
*q = 55; // oops!
```



- Example 2

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

Destroyed when function exits!! (points to `char buffer[80];`)

3/25/2001 J-28

Anything Wrong?

```
void swap (book & a, book & b) {
  book * temp;
  *temp = a;
  a = b;
  b = *temp;
}

// example call:
swap(book1, book2);  // note:  no &
```

3/25/2001 J-29

Security Crack

```
int * Performance::getDuration (void) {
        return  &duration; //duration is a private
                         //member variable of the class
}

//client
performance perf1;
....
int *  dur = perf1.getDuration( );
```

3/25/2001 J-30

Giving Away What's Not Yours

Performance X ("Pearl Jam", "Main Stage");
Performance * Y = &X; //OK
Y -> setTime (3, 30, 70); //OK
...
delete Y; //don't do it!

3/25/2001 J-31

new with Classes

- If the object that you allocate with *new* is a class instance: *then the constructor has been called.*
  - Might be the default constructor

```
bankAccount *BP;  //no constructor called here!
BP = new bankAccount; //constructor called
bankAccount * AllAccounts = new bankAccount[1000];
//Reminder: system-supplied default does not initialize member variables
```

- You can pass arguments to constructors, too.

```
bankAccount * b1 = new bankAccount ("J. Smith", 5.00);
```

- What's wrong with this one?

bankAccount BadB = new bankAccount;

3/25/2001 J-32

Safety Guidelines

- Avoid creating garbage when invoking **new** or moving pointers.
- Don't lose the pointer
- Don't dereference an unassigned pointer.
- After **new**, check that the pointer is not NULL
- After **delete**, don't use the pointer again
  - If paranoid, set the pointer to NULL yourself
- Avoid security cracks

3/25/2001 J-33

Detour: Arrays vs. Pointers

- An array name refers to the address of the first element of the array

char qarr [10]; //true or false: qarr == & (qarr[0])

- Array notation can be used with pointers, and vice-versa

```
bool manglestring (char aName[], char * bName) {
    int i = 0;
    while (bName[i] != '\0'){
        aName[i] = bName[i];
        i++;
    }
    aName[i] = '\0';
    if (islower (*aName)){
        ...
    }
```

3/25/2001 J-34

"Dynamic" Arrays

- We can get "dynamic" arrays this way
- Old "static" arrays:

const int MAX_BOOKS = 20;
book bookArray[MAX_BOOKS];

- New "dynamic" arrays:

int book_count = 20;
*book *bookArray = new book[book_count];*
...
*book_count = 2 * book_count;*
//this does <u>not</u> change the size of bookArray!!

3/25/2001 J-35

Nevertheless... Arrays ≠ Pointers!

```
int * ip;            //what memory is allocated?
int iarr[10];        //what memory is allocated?

iarr[0] = 100;       //good or bad?
ip[0] = 200;         //good or bad?
ip = iarr;           //good or bad?
iarr = ip;           //good or bad?
ip = new int[20];    //good or bad?
iarr = new int[20];//good or bad?
```

3/25/2001 J-36

Guru Stuff: Pointer Arithmetic

- You can do arithmetic on pointers
- p+1 points to the next item of its type
  - Does *not* mean "the next byte after p"
  - Takes into account the size of the type
- Under the hood:
  - Arr[N] is really *(Arr + N)

3/25/2001 J-37

Trace and Find Mem. Errors

```
int *p1, *p2;      // line 1
int i;             // line 2
p1 = new int;      // line 3
*p2 = 5;           // line 4
int *p3 = p1;      // line 5
p2 = new int[4];   // line 6
delete p3;         // line 7
p3 = NULL;         // line 8
p2 = &i;           // line 9
*p1 = 15;          // line 10
delete p2;         // line 11
```

3/25/2001 J-38