ICOM 4035 – Data Structures

Dr. Manuel Rodríguez Martínez
Electrical and Computer Engineering Department
Lecture 3 – August 28, 2001

Readings

- Read chapter 1 of textbook
  - Arrays, pointers and structures
    - Do not read section 1.6.3

More on pointers

- Suppose we have the declaration

  int x;

  – Then the following is a legal pointer declaration;

  int *ptr = &x;

  – Now ptr points to the memory location allocated for variable x.

  – The following is illegal (compiler error!):

  int *ptr = x;

  – Variable x is not a pointer, its value is an integer number not a memory location. C++ will enforce it. C won't.

  – The following is semantically incorrect but C++ compiler won't catch it

  int ptr;

  *ptr = x;

  – ptr has no memory allocated for it!

Pointer de-reference

- The * gives the contents of the memory location pointed to by a pointer.
- It is a way to access the storage area
  - Behaves like a regular variable of the type associated with the pointer.
  - Example:

```
int x= 2;
int *ptr1 = &x, ptr2 = NULL;
*ptr1 = 3; // now x becomes 3
ptr2 = new int(10);
cout << (*ptr1); // prints out 3
cout << (*ptr2); // prints out 10
```

Pointers and operator precedence

- Must consider operator precedence when using pointers.
  - Example 1:

    int ptr = new int(10);

    *ptr += 20;

    - This is equivalent to:

    *ptr = *ptr + 20;

    - * has higher precedence than +=
  - Example 2:

    int *ptr = new int(10);

    *ptr++;

    - This is equivalent to:

    ptr++; // change pointer address!!!

    *ptr; // might give run time error

    - ++ has higher precedence than *

Pointers and arrays

- A built-in array is just a pointer!
- These are equivalent:

```
int nums1[5] ;  // arrays of 10 elements

int *nums2=NULL;
nums2 = new int[5]; // array of 10 elements;
```

- Same access patterns:

```
for (int i=0; i < n; ++i){
    nums1[i] = 1;
    nums2[i] = 2;
}
```

Dynamic Memory Allocation

- Local variables and parameters used in functions are cleanup by the run-time system.
- Memory is allocated using the new operator
  - This memory space will not be cleanup automatically by the run-time.
  - If we forget to “recycle” unused memory space, we get memory leaks.
    - Memory space that cannot be used. It is basically wasted!
    - Programs can crash due to lack of memory associated with memory leaks.
- Delete operator is used to “recycle” memory space.
  - Apply it to pointer variables

Example of memory allocation

- Add random numbers

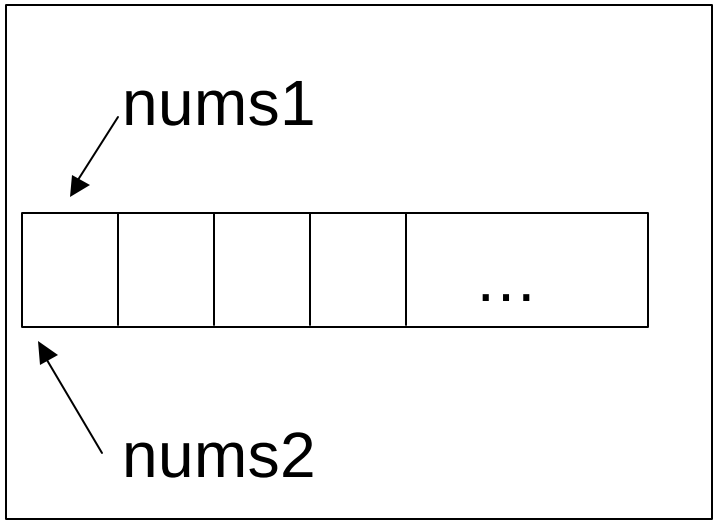
```
int *nums = NULL;
int size = 10;
int seed = random(100); // random number between 0 and 100
for (int i=0; i < size; ++i){
    nums[i] = seed++;
}
cout << sum(nums);
delete [] nums;
```

- Apply delete to single value or arrays:
  - Single values: delete ptr;
  - Array values: delete [] nums;

Stale pointer problem

- Consider the following:

```
int *nums1=NULL, *nums2 = NULL;
int size = 100, i=0;
nums1 = new int[size];
nums2 = nums1;
for (i=0; i < size; ++i){
    nums1[I] = i;
}
delete [] nums2;
cout << nums1[0]; // should be run-time error (not in g++ ???!!!)
```



  - By deleteing nums2, we also deleted nums1 (became stale)
    - Common problem when a pointer parameter is “accidentally” recycled with delete

Pointers to structures

- Consider the following structure

```
typedef struct student {
    string name;  // name is an object
    int age;
}student;
```

  - We can declare a pointer to student struct as follow

```
student *std = NULL;
std = new student;
```

- To access the individual fields you use -> operator:
  - std->name = “Jose”; // access to name field
  - std->age = 25; // access to age field
  - Alternative is annoying:
    - (*std).name = “Jose”; // get contents of pointer, then use dot

Initializing fields that are pointers

- If you get a structure with pointers in it, you MUST allocate and initialize these fields.
- Example 1:

```
typedef struct row {
    int size;
    int *columns;
} row;
row theRow ;
theRow.size = 2;
theRow.columns = new int [2];
theRow.columns[0] = 1;
theRow.columns[1] = 2;
```

Initializing fields that are pointers

- If the you get a pointer to a structure, and the structures has pointers in it, you MUST allocate all these pointers
- Example:

```
typedef struct row {
    int size;
    int *columns;
} row;
row *theRow;
theRow = new row;
theRow->size = 2;
theRow->columns = new int [2];
theRow->columns[0] = 1;
theRow->columns[1] = 2;
```

Memory Alignment Problem

- Remember that CPU must access memory based on word boundaries.
- Suppose your computer has a 32-bit architecture.
- Consider the following declaration:

```
typedef struct record{
        int num;  // 4-byte int
        char letter; // 1-byte char
} record;
```

  - Structure has a 4 byte int field and a 1 byte char field.
  - But structure size is 8 bytes!
    - char field must be aligned to a 4-byte word
    - Always used the sizeof() operator to estimate size of structs and other objects!

Memory Alignment Problem

```
typedef struct record{
        int num;  // 4-byte int
        char letter; // 1-byte char
}record;
```