Chapter 10
Pointers & Dynamic Arrays

Pointer Definition

• A pointer contains the memory address of a variable
• You’ve used pointers already
  - Call-by-reference parameters
    • Address of actual argument was passed
• Example:
  
  double *p;
  
  - p is declared a "pointer to double" variable
  - Can hold pointers to variables of type double
    • Not other types
Declaring Pointers

- Pointers declared like other types
  - Add "*" before variable name
  - Produces "pointer to" that type
- "*" must be before each variable
- int *p1, *p2, v1, v2;
  - p1, p2 hold pointers to int variables
  - v1, v2 are ordinary int variables

Address of operator

- int *p1, *p2, v1, v2;
  p1 = &v1;
  - Sets pointer variable p1 to "point to" int variable v1
- Operator, &
  - Determines "address of" variable
  - Same operator used in call-by-reference
- Read like:
  - "p1 points to v1"
Dereferencing Pointers

• Recall:
  int *p1, *p2, v1, v2;
  p1 = &v1;

• Two ways to refer to v1 now:
  – Variable v1 itself:
    cout << v1;
  – Via pointer p1:
    cout << *p1;

• Dereference operator, *
  – Pointer variable "de-referenced"
  – Means: "Get data that p1 points to"

Assigning Pointers

• Pointer variables can be "assigned":
  int *p1, *p2;
  p2 = p1;
  – Assigns one pointer to another
  – "Make p2 point to where p1 points"

• Do not confuse with:
  *p1 = *p2;
  – Assigns "value pointed to" by p1, to "value pointed to" by p2
Assignment Example

Display 10.1  Uses of the Assignment Operator with Pointer Variables

p1 = p2;

*p1 = *p2;

New operator

• Can dynamically allocate variables
  - Operator new creates variables
    • No identifiers to refer to them
    • Just a pointer
  
• p1 = new int;
  - Creates new "nameless" variable, and assigns p1 to "point to" it
  - Can access with *p1
    • Use just like ordinary variable
New Operator Example

More on new operator

- Creates new dynamic variable
- Returns pointer to the new variable
- If type is class type:
  - Constructor is called for new object
  - Can invoke different constructor with initializer arguments:

  ```
  MyClass *mcPtr;
  mcPtr = new MyClass(32.0, 17);
  ```

- Can still initialize non-class types:

  ```
  int *n;
  n = new int(17); //Initializes *n to 17
  ```
Pointers and Functions

- Can be function parameters
- Can be returned from functions
- This function declaration:

  ```c
  int* findOtherPointer(int* p);
  - Has "pointer to an int" parameter
  - Returns "pointer to an int" variable
  ```

Memory Management

- Heap
  - Also called "freestore"
  - Reserved for dynamically-allocated variables
  - All new dynamic variables consume memory in freestore
    - If too many → could use all freestore memory
- Future "new" operations will fail if freestore is "full"
Checking if new succeeded

• Older compilers:
  – Test if null returned by call to new:

    ```cpp
    int *p = new int;
    if (p == NULL) {
        cout << "Aborting. Insufficient memory.\n";
        exit(1);
    }
    ```

• Newer compilers:
  – If new operation fails:
    • Program terminates automatically
    • Produces error message
  – Still good practice to use NULL check

Freestore Size

• Varies with implementations
• Typically large
  – Most programs won’t use all memory
• Memory management
  – Still good practice
  – Solid software engineering principle
  – Memory IS finite
    • Regardless of how much there is
Delete Operator

- De-allocate dynamic memory
  - When no longer needed
  - Returns memory to freestore
  - Example:

    ```
    int *p;
    p = new int(5);
    ... //Some processing...
    delete p;
    ```

  - De-allocates dynamic memory "pointed to by pointer p"

Dangling Pointer

- delete p;
  - Destroys dynamic memory
  - But p still points there!
    - Called "dangling pointer"
  - If p is then dereferenced ( *p )
    - Unpredictable results!
    - Often disastrous!

- Avoid dangling pointers
  - Assign pointer to NULL after delete:

    ```
    delete p;
    p = NULL;
    ```
Dynamic & Automatic Variables

• Dynamic variables
  – Created with new operator
  – Created and destroyed while program runs

• Local variables
  – Declared within function definition
  – Not dynamic
    • Created when function is called
    • Destroyed when function call completes
  – Often called "automatic" variables
    • Properties controlled for you

Static vs Dynamic Arrays

• Static arrays
  – int nums[SIZE]
  – Size is known at compile time
  – Fixed size

• Dynamic Arrays
  – int* nums = new int[SIZE]
  – Size is determined at run-time
  – Can grow and shrink as needed
Array Variables

• Arrays are stored sequentially in memory
  – Name of array is address of first element
  – Therefore the name of an array is a pointer

• Example:
  ```
  int a[10];
  int* p;
  ```
  – a and p are both pointers so we can perform assignments
  ```
  p = a;  // p now points to array a
  a = p;  // ILLEGAL: a is a constant pointer i.e
           // const int* type
  ```

Dynamic Arrays

• Estimate initial size and allocate that amt of memory
  ```
  int* nums = new int[size];
  ```

• When array is full create a bigger one and copy elements
  ```
  int* temp = new int[size + increment];
  for (int i = 0; i < size; ++i)
    temp[i] = nums[i];
  ```

• Delete old array to avoid memory leaks
  ```
  delete [] nums;  // Notice[]. Means delete array.
  ```

• Reassign pointer so new array has same name as old
  ```
  nums = temp;
  ```

• Update array size
  ```
  size += increment;
  ```
Returning Arrays

- Array type not allowed as return-type of function

- Example:
  ```c
  int [] someFunction();  // Illegal
  ```

- Instead return pointer to array base type:
  ```c
  int* someFunction();  // Legal
  ```

- This means that the function that created the array is not responsible for deleting it

Pointer Arithmetic

- Can perform arithmetic on pointers
  - "Address" arithmetic

- Example:
  ```c
  double* d = new double[10];
  ```

  - d contains address of d[0]
  - d + 1 evaluates to address of d[1]
  - d + 2 evaluates to address of d[2]
    - Equates to "address" at these locations
Array Manipulation

- Can access elements of an array using
  - Pointer arithmetic
    ```cpp
    for (int i = 0; i < arraySize; ++i)
        cout << *(d + i) << endl;
    // OR
    int i = 0;
    while(i < arraySize) {
        cout << *d << endl;
        d++;
        i++;
    }
  ```
  - Regular array indexing
    ```cpp
    for (int i = 0; i < arraySize; i++)
        cout << d[i] << endl;
    ```
- Only addition/subtraction on pointers
  - No multiplication, division

Multi-dimensional Arrays

- Create an "array of arrays"
- To create a 4x3 array:
  - Create an array of 4 pointers (rows)
    ```cpp
    float** float_values = new float*[4];
    ```
  - Allocate an array of 3 elements to each row pointer
    ```cpp
    for (int i = 0; i < 4; i++)
        float_values[i] = new float[3];
    ```
Arrow Operator

• The -> operator
  - Shorthand notation
• Combines dereference operator * and dot operator
• Example:

  ```cpp
  MyClass *p = new MyClass;
p->print();
  ```

• Equivalent to:

  ```cpp
  (*p).print();
  ```

This pointer

• Member function definitions might need to refer to calling object
• Use predefined **this** pointer
  - Automatically points to calling object:
    ```cpp
    class Simple {
      public:
        void showStuff() const;
      private:
        int stuff;
    };
    ```
• Two ways for member functions to access:

  ```cpp
  cout << stuff;
cout << this->stuff;
  ```
Destructors

• Opposite of constructor
  – Automatically called when object is out-of-scope
  – Default version only removes static variables
• Must manually delete dynamically-allocated variables
• Defined like constructor, just add ~

```cpp
MyClass::~MyClass() {
    //Delete all dynamically allocated variables
}
```