Const Pointers as Function Parameters

Four ways to use const with pointers and functions
- Non-constant pointer to non-constant data
  - Data can be manipulated and pointer can change
- Non-constant pointer to constant data
  - Pointer can be made to point to something else, data cannot change
- Constant pointer to non-constant data
  - Data can be changed, pointer cannot
- Constant pointer to constant data
  - Nothing can change

Simple, Right?

Array Pointer

- Assume we have int b[6] and int * bPtr
- We can do this:
  - bPtr = b;
  - bPtr = &b[ 0 ];
- Also, for example b[3] is:
  - ( bPtr + 3 )
  - *( b + 3 )

Deallocation

Failure to explicitly delete a dynamic variable will result in that memory NOT being returned to the system, even if the pointer to it goes out of scope.
- This is called a "memory leak" and is evidence of poor program implementation.
- If large dynamic structures are used (or lots of little ones), a memory leak can result in depletion of available memory.

Pointers Passed

- Passed by value
  - When they do not need to change the pointer value itself
- Passed by reference
  - When they change what the pointer is pointing to

Dynamic Memory Problems

Garbage
- Previously allocated memory that is inaccessible thru any program pointers or structures.
- Example:

```
before
iptr1 = new int (6);
*iptr1 = NULL;
during
after
```

Dynamic Memory Problems

- **Aliases**
  - Two or more pointers referencing the same memory location.
  - Example:
    ```
    iptr1 = new int (6);
    iptr2 = iptr1;
    ```

- **Dangling Pointers**
  - Pointers that reference memory locations previously deallocated.
  - Example:
    ```
    iptr1 = new int (6);
    iptr2 = iptr1;
    delete iptr1;
    ```

Copy Constructor

- A copy constructor allows you to successfully create an object that is a copy of another
- e.g. `Student NewStudent = OldStudent;`
- This would invoke the copy constructor.
- The copy constructor would take care of creating and copying the course information

Assignment Operator

- An assignment operator allows you to transfer a copy of an already existing object into an already existing object.
- e.g. `StudentA = StudentB;`
- This is a simple assignment statement.
- The difference between this and a copy constructor is the missing `StudentA` already exists

Reference Variable

- The ampersand `&` character is used for reference variable declarations:
  ```
  int & iptr;
  float & fptr1, & fptr2;
  ```
- Reference variables do NOT use the address and dereference operators (`& *`).
- Compiler dereferences reference variables transparently.
Stacks

- Restricted list structure
  - Dynamic LIFO Storage Structure
    † Size and Contents can change during execution of program
    † Last In First Out (lifo)
- Elements are added to the top and removed from the top
- How do you implement one?
  - What about a dynamic array?
  - What about a linked list?
  - What about a string?

Queues

- Restricted (two-tailed) list structure
- Dynamic FIFO Storage Structure
  - Size and Contents can change during execution of program
  - First in First Out
  - Elements are inserted (enqueue) into the rear and retrieved (dequeue) from front.
- Think of waiting in line to check-out of a store.

Whaaaaaat?^

- Code operations to force array indicies to ‘wrap-around’
  † front = (front) % MAXQUE ;
  † rear = (rear) % MAXQUE ;

Array Math

- Distinct States
  † Full Queue: (que.rear + 1) % MAXQUE == que.front
  † Empty Queue: (que.rear == que.front )
  † One-element Queue: (que.front + 1) % MAXQUE == que.rear

DOS

- “Double-Ended” Queue
  - variation of a regular queue.
  - elements can be added and removed at either the rear or front of the queue, but nowhere else in the queue.
  - operations:
    Deque(), Empty(), Full(), EnqRear(), EnqFront(), DeqFront(), DeqRear()
  - generalization of both a stack and a queue.
Namespaces

A namespace is a scope with a name attached. That is:

```c
namespace FooSpace {
    typedef struct {
        string Message;
        int Target;
    } Foo;
    const int MaxFoo = 1000;
    int numFoo;
    Foo List[MaxFoo];
};
```