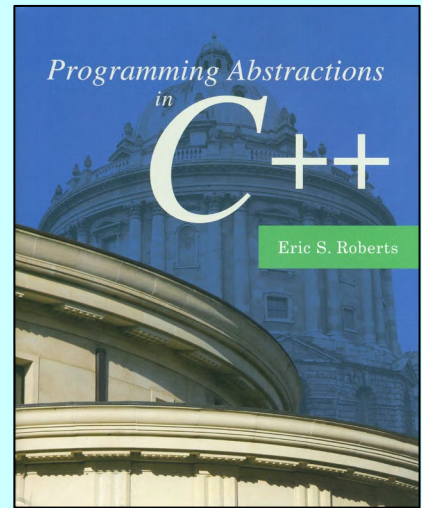


CHAPTER 14

Linear Structures

It does not come to me in quite so direct a line as that; it takes a bend or two, but nothing of consequence.

—Jane Austen, *Persuasion*, 1818



14.0 Implementing character stacks revisited

14.1 Templates

14.2 Implementing stacks

14.3 Implementing queues

14.4 Implementing vectors

14.5 Integrating prototypes and code

Example: The CharStack Class

- Write classes that use dynamic allocation
- **CharStack**: a stack of characters

CharStack cstk;

Initializes an empty stack.

cszk.size()

Returns the number of characters pushed onto the stack.

cszk.isEmpty()

Returns **true** if the stack is empty.

cszk.clear()

Deletes all characters from the stack.

cszk.push(ch)

Pushes a new character onto the stack.

cszk.pop()

Removes and returns the top character from the stack.

The charstack.cpp Implementation

```
class CharStack {
...
private:
    char *array;           /* Dynamic array of characters */
    int capacity;          /* Allocated size of that array */
    int count;             /* Current count of chars pushed */
...
}

void CharStack::push(char ch) {
    if (count == capacity) expandCapacity();
    array[count++] = ch;
}

void CharStack::expandCapacity() {
    char *oldArray = array;
    capacity *= 2;
    array = new char[capacity];
    for (int i = 0; i < count; i++) {
        array[i] = oldArray[i];
    }
    delete[] oldArray;
}
```

Exercise: The **IntStack** Class

- Can you write a class **IntStack**: a stack of integers?
- What part in your **CharStack** code needs change, and what remains the same?
- How about a stack of Booleans, floating-point numbers, etc.?
- How about a stack of enumerated types, e.g., **Direction**?
- How about a stack of structures, pointers, arrays, etc.?
- How about a stack of a user-defined data types, e.g., **Point**?
- How about a stack for every data type?

Overloading

- In programming languages, *polymorphism* is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types.
- Function/operator *overloading* (a form of polymorphism) allows you to define several functions with the same name as long as those functions can be distinguished by their signatures.
- Given a particular function call, the compiler can look at the number and types of the arguments (i.e., signature) and choose the version of the function that matches that signature.
- Overloaded functions can have quite different behaviors with different signatures; or simply different parameter types, but with the exact same body, which is exactly what we need to do to **CharStack** to implement the new **Stack** class.

Exercise: Overload

For this type of overloading, C++ can help you generate the overloading automatically.

- What is the output?

```
#include <iostream>
using namespace std;

int operate(int a, int b)
{
    return a*b;
}

double operate(double a, double b)
{
    return a/b;
}

int main ()
{
    cout<<operate(1, 2)<<endl;
    cout<<operate(1.0, 2.0)<<endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;

int operate(int a, int b)
{
    return a/b;
}

double operate(double a, double b)
{
    return a/b;
}

int main ()
{
    cout<<operate(1, 2)<<endl;
    cout<<operate(1.0, 2.0)<<endl;
    return 0;
}
```



Templates

- One of the most powerful features in C++ is the **template** facility, which makes it possible to define functions and classes that work for a variety of types, e.g., **to automate the overloading process in situations where the code is identical except for the types involved.**
- The most common form of a template specification is

```
template <typename placeholder>
```

where *placeholder* is an identifier that is used to stand for a specific type when the definition following the **template** specification is compiled. The keyword **typename** can also be **class**. They are **interchangeable in this context.**

- For more than one type parameter, simply specify more template parameters between the angle brackets.

```
template <typename placeholder1, typename placeholder2>
```



Templates in Functions

- Templates can be used before a function definition to create a generic collection of functions that can be applied to values of various types.
- The following code, for example, creates a template for the **max** function, which returns the larger of its two arguments:

```
template <typename ValueType>
ValueType max(ValueType v1, ValueType v2) {
    return (v1 > v2) ? v1 : v2;
}
```

- The compiler will generate the code for many different versions of **max**, one for each type that the client uses.
- Note that a template is not a class or a function. It is a “pattern” that the compiler uses to generate a family of classes or functions.

Templates in Functions

- The function **max** can be used only with types that implement the **>** operator. If you call **max** on some type that doesn't, the compiler will signal an error.

```
max('A', 'Z');
```

```
char * max(char * v1, char * v2) {  
    return (v1 > v2) ? v1 : v2;  
} // unspecified behavior
```

```
max("cat", "dog");
```

error: call of overloaded
'max(std::string, std::string)'
is **ambiguous**

```
#include <string>  
max(std::string("cat"), std::string("dog"));
```

```
#include <string>  
std::max(std::string("cat"), std::string("dog"));
```

```
#include <string>  
::max(std::string("cat"), std::string("dog"));
```

Exercise: Multiple Types

- The following code creates a template for the **max** function, which returns the larger of its two arguments:

```
template <typename ValueType1, typename ValueType2>
ValueType1 max(ValueType1 v1, ValueType2 v2) {
    return (v1 > v2) ? v1 : v2;
}
```

- What is the result of the following function call and why?

```
max(1, 1.1);
```

Multiple Types in Modern C++

- C++14

```
template <typename ValueType1, typename ValueType2>  
auto max(ValueType1 v1, ValueType2 v2) {  
    return (v1 > v2) ? v1 : v2;  
}
```

- C++20

```
auto max(auto v1, auto v2) {  
    return (v1 > v2) ? v1 : v2;  
}
```

Things that you only need to know that you don't know. I don't need you to memorize it for the exam.

Example: Rewrite `sort` as a Template

- Templates can be used before a function definition to create a generic collection of functions that can be applied to values of various types.
- Rewrite the following code so that it sorts any type that implements the `<` operator:

```
void sort(int array[], int n) {
    for (int i = 0; i < n; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (array[j] < array[minIndex]) minIndex = j;
        }
        swap(array[i], array[minIndex]);
    }
}

void swap(int & x, int & y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```

Templates in Functions

- The following code, for example, creates a template for invoking selection sort on arrays of any element type:

```
template <typename ValueType>
void sort(ValueType array[], int n) {
    for (int i = 0; i < n; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (array[j] < array[minIndex]) minIndex = j;
        }
        swap(array[i], array[minIndex]);
    }
}

template <typename ValueType>
void swap(ValueType & x, ValueType & y) {
    ValueType tmp = x;
    x = y;
    y = tmp;
}
```

Where Do We Go From Here?

- Our goal from this point is to **implement the (Stanford) collection classes using templates** to support different base types, and to compare the algorithmic efficiency of the various options.
- We are going to use low-level data structures, such as **arrays** and **linked lists**, which support **dynamic allocation**.
- We will start with the linear collection classes, such as **Stack**, **Queue**, **Vector**, etc.

A Template Version of the **Stack** Class

- The first step in writing the template version of the **Stack** class is to add the **template** keyword to the interface just before the class definition:

```
template <typename ValueType>
class Stack {
    . . . body of the class . . .
};
```

- Once you have made this change, each instance of the specific type (formerly **char** in the **CharStack** class) must be replaced by the **ValueType** placeholder for the generic type, as in

```
ValueType *array;
```

or

```
void push(ValueType value);
```

Implementing the Template Class

- The final change necessary to implement the template class is to add **template** declarations to every method body, as in the following updated version of the constructor:

```
template <typename ValueType>
Stack<ValueType>::Stack() {
    capacity = INITIAL_CAPACITY;
    array = new ValueType[capacity];
    count = 0;
}
```

To instantiate a template class, you specify arguments for the template class enclosed by angle brackets after the class name wherever the class is used.

- Because of the restrictions that C++ imposes on template types, **the implementations of the methods need to be included as part of the `stack.h` header**, even though the details are not of interest to most clients.



Templates in Class Definitions

- Templates are more commonly used to define **generic** classes. When they are used in this way, the **template** keyword must appear before the class definition and before each of the implementations of the member functions.
- The most **inconvenient** aspect of using templates to create generic classes is that the compiler cannot process them correctly unless it has access to both the interface and the implementation at the same time. The effect of this restriction is that **the .h files for template classes must contain *both* the prototypes and the corresponding code.**
- To emphasize the conceptual separation between the interface and the associated implementation, you should make sure to include an appropriate comment before the private section and the implementation in the .h files warning casual clients away from the details.

The `stack.h` Interface

```
/*
 * File: stack.h
 * -----
 * This interface exports a template version of the Stack class. This
 * implementation uses a dynamic array to store the elements of the stack.
 */

#ifndef _stack_h
#define _stack_h

#include "error.h"

/*
 * Class: Stack<ValueType>
 * -----
 * This class implements a stack of the specified value type.
 */

template <typename ValueType>
class Stack {

public:
```

The `stack.h` Interface

```
/*
 * Constructor: Stack
 * Usage: Stack<int> stack;
 * -----
 * The constructor initializes a new empty stack containing
 * the specified value type.
 */
    Stack();

/*
 * Destructor: ~Stack
 * Usage: (usually implicit)
 * -----
 * The destructor deallocates any heap storage associated
 * with this stack.
 */
    ~Stack();
```

The `stack.h` Interface

```
/* Comments for the simple methods have been elided for space */
    int size() const;

/* . . . */
    bool isEmpty() const;

/* . . . */
    void clear();

/* . . . */
    void push(ValueType value);

/* . . . */
    ValueType pop();

/* . . . */
    ValueType peek() const;

/* Copy constructor and assignment operator */
    Stack(const Stack<ValueType> & src);
    Stack<ValueType> & operator=(const Stack<ValueType> & src);
```

The `stack.h` Interface

```
/* Private section */

/*
 * Implementation notes
 * -----
 * This version of the stack.h interface uses a dynamic array to store
 * the elements of the stack. The array begins with INITIAL_CAPACITY
 * elements and doubles the size whenever it runs out of space. This
 * discipline guarantees that the push method has  $O(1)$  amortized cost.
 */

private:

    static const int INITIAL_CAPACITY = 10;

/* Instance variables */

    ValueType *array;           /* A dynamic array of the elements */
    int capacity;               /* The allocated size of the array */
    int count;                  /* The number of stack elements */

/* Private method prototypes */

    void deepCopy(const Stack<ValueType> & src);
    void expandCapacity();

};
```

The `stack.h` Interface

```
/*
 * Implementation section
 * -----
 * C++ requires that the implementation for a template class be available
 * to the compiler whenever that type is used. The effect of this
 * restriction is that header files must include the implementation.
 * Clients should not need to look at any of the code beyond this point.
 */

/*
 * Implementation notes: Stack constructor and destructor
 * -----
 * These methods allocate and free the dynamic array.
 */

template <typename ValueType>
Stack<ValueType>::Stack() {
    capacity = INITIAL_CAPACITY;
    array = new ValueType[capacity];
    count = 0;
}

template <typename ValueType>
Stack<ValueType>::~~Stack() {
    delete[] array;
}
```

The `stack.h` Interface

```
/*
 * Implementation notes: size, isEmpty, clear
 * -----
 * These methods are generic forms of the versions used for CharStack.
 */

template <typename ValueType>
int Stack<ValueType>::size() const {
    return count;
}

template <typename ValueType>
bool Stack<ValueType>::isEmpty() const {
    return count == 0;
}

template <typename ValueType>
void Stack<ValueType>::clear() {
    count = 0;
}
```

The `stack.h` Interface

```
/*
 * Implementation notes: push, pop, peek
 * -----
 * The push method tests to see whether the array needs to be expanded; pop
 * and peek must check for an empty stack.
 */

template <typename ValueType>
void Stack<ValueType>::push(ValueType value) {
    if (count == capacity) expandCapacity();
    array[count++] = value;
}

template <typename ValueType>
ValueType Stack<ValueType>::pop() {
    if (isEmpty()) error("pop: Attempting to pop an empty stack");
    return array[--count];
}

template <typename ValueType>
ValueType Stack<ValueType>::peek() const {
    if (isEmpty()) error("peek: Attempting to peek at an empty stack");
    return array[count - 1];
}
```


The `stack.h` Interface

```
/*
 * Implementation notes: copy constructor and assignment operator
 * -----
 * These methods follow the standard template, leaving the work to deepCopy.
 */

template <typename ValueType>
Stack<ValueType>::Stack(const Stack<ValueType> & src) {
    deepCopy(src);
}

template <typename ValueType>
Stack<ValueType> & Stack<ValueType>::operator=(const Stack<ValueType> & src) {
    if (this != &src) {
        delete[] array;
        deepCopy(src);
    }
    return *this;
}
```

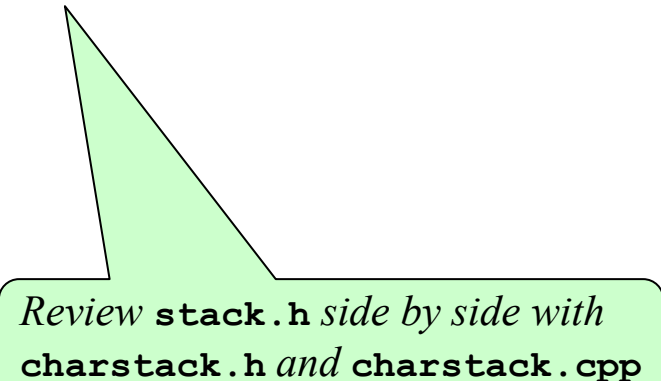
The `stack.h` Interface

```
/*
 * Implementation notes: deepCopy
 * -----
 * This function copies the data from the src parameter into the current
 * object. All dynamic memory is reallocated to create a "deep copy" in
 * which the current object and the source object are independent.
 * The capacity is set so that the stack has some room to expand.
 */

template <typename ValueType>
void Stack<ValueType>::deepCopy(const Stack<ValueType> & src) {
    capacity = src.count + INITIAL_CAPACITY;
    this->array = new ValueType[capacity];
    for (int i = 0; i < src.count; i++) {
        array[i] = src.array[i];
    }
    count = src.count;
}
```

The `stack.h` Interface

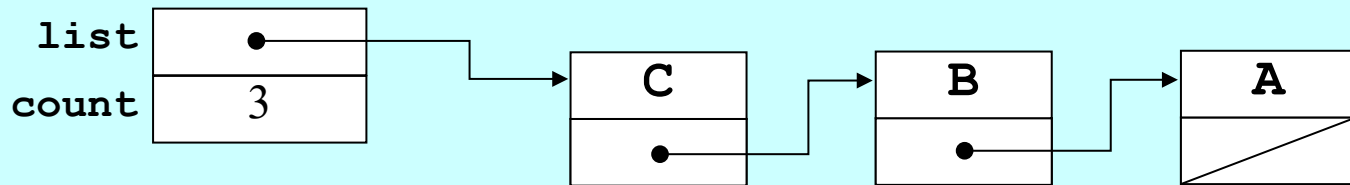
```
/*  
 * Implementation notes: expandCapacity  
 * -----  
 * This private method doubles the capacity of the elements array whenever  
 * it runs out of space. To do so, it copies the pointer to the old array,  
 * allocates a new array with twice the capacity, copies the values from  
 * the old array to the new one, and finally frees the old storage.  
 */  
  
template <typename ValueType>  
void Stack<ValueType>::expandCapacity() {  
    ValueType *oldArray = array;  
    capacity *= 2;  
    array = new ValueType[capacity];  
    for (int i = 0; i < count; i++) {  
        array[i] = oldArray[i];  
    }  
    delete[] oldArray;  
}  
  
#endif
```



*Review `stack.h` side by side with
`charstack.h` and `charstack.cpp`*

Implementing Stacks Using Lists

- It is also possible to implement stacks using the list structure. In this model, the top of the stack is the first element in the list and each additional item appears one cell further down. A stack in which the values 'A', 'B', and 'C' have been pushed in that order therefore looks like this:



- Can we reverse the order**, i.e., the bottom of the stack is the first element in the list, and why?
- The implementation of the *list-based stack* is omitted here because it is much simpler than the corresponding code for the *list-based queue*, which appears in full later.
- Why is the *list-based stack* simpler than the *list-based queue*?**

Methods in the Queue<x> Class

queue.size()

Returns the number of values in the queue.

queue.isEmpty()

Returns **true** if the queue is empty.

queue.enqueue(value)

Adds a new value to the end of the queue (which is called its *tail*).

queue.dequeue()

Removes and returns the value at the front of the queue (which is called its *head*).

queue.peek()

Returns the value at the head of the queue without removing it.

queue.clear()

Removes all values from the queue.

The queue.h Interface

```
/*
 * File: queue.h
 * -----
 * This interface defines a general queue abstraction that uses
 * templates so that it can work with any element type.
 */

#ifndef _queue_h
#define _queue_h

/*
 * Template class: Queue<ValueType>
 * -----
 * This class template models a queue, which is a linear collection
 * of values that resemble a waiting line. Values are added at
 * one end of the queue and removed from the other. The fundamental
 * operations are enqueue (add to the tail of the queue) and dequeue
 * (remove from the head of the queue). Because a queue preserves
 * the order of the elements, the first value enqueued is the first
 * value dequeued. Queues therefore operate in a first-in-first-out
 * (FIFO) order. For maximum generality, the Queue class is defined
 * using a template that allows the client to define a queue that
 * contains any type of value, as in Queue<string> or Queue<taskT>.
 */
```

The queue.h Interface

```
template <typename ValueType>
class Queue {
public:
    /*
    * Constructor: Queue
    * Usage: Queue<ValueType> queue;
    * -----
    * Initializes a new empty queue containing the specified value type.
    */
    Queue () ;

    /*
    * Destructor: ~Queue
    * -----
    * Deallocates any heap storage associated with this queue.
    */
    ~Queue () ;
```

The queue.h Interface

```
/* Comments for the simple methods have been elided for space */
    int size();
/* . . . */
    bool isEmpty();
/* . . . */
    void clear();
/* . . . */
    void enqueue(ValueType value);
/* . . . */
    ValueType dequeue();
/* . . . */
    ValueType peek();

/* Copy constructor and assignment operator */
    Queue(const Queue<ValueType> & src);
    Queue<ValueType> & operator=(const Queue<ValueType> & src);

/* The implementation of copy constructor and assignment operator,
 * and hence deepCopy will be elided in later slides for space */
```


The queue.h Interface

```
/* Private section */
```

Private section goes here.

```
};
```

```
/*
```

```
* The template feature of C++ works correctly only if the compiler  
* has access to both the interface and the implementation at the  
* same time. As a result, the compiler must see the code for  
* the implementation at this point, even though that code is  
* not something that the client needs to see in the interface.  
*/
```

Implementation section goes here.

```
#endif
```



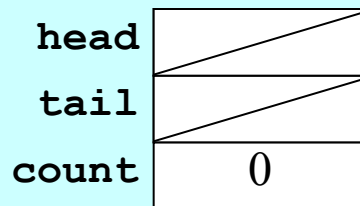
Implementing a Linked-List Queue

- In some ways, the linked-list implementation of a queue is easier to understand than the array-based model, even though it contains pointers.
- In the linked-list version, the private data structure for the **Queue** class requires two pointers to cells: a **head** pointer that indicates the first cell in the chain, and a **tail** pointer that indicates the last cell.
- The **enqueue** operation adds a new cell after the one marked by the **tail** pointer and then updates the **tail** pointer so that it continues to indicate the end of the list. The **dequeue** operation consists of removing the cell addressed by the **head** pointer and returning the value in that cell.
- Can we **enqueue** at the head and ~~**dequeue** at the tail?~~
- The private data structure must also **keep track of the number of elements** so that the **size** method can run in $O(1)$ time.

Tracing the List-Based Queue

```
→ Queue<char> queue;  
→ queue.enqueue('A')  
   queue.enqueue('B')  
   queue.dequeue()  
   queue.enqueue('C')  
   queue.dequeue()  
   queue.dequeue()
```

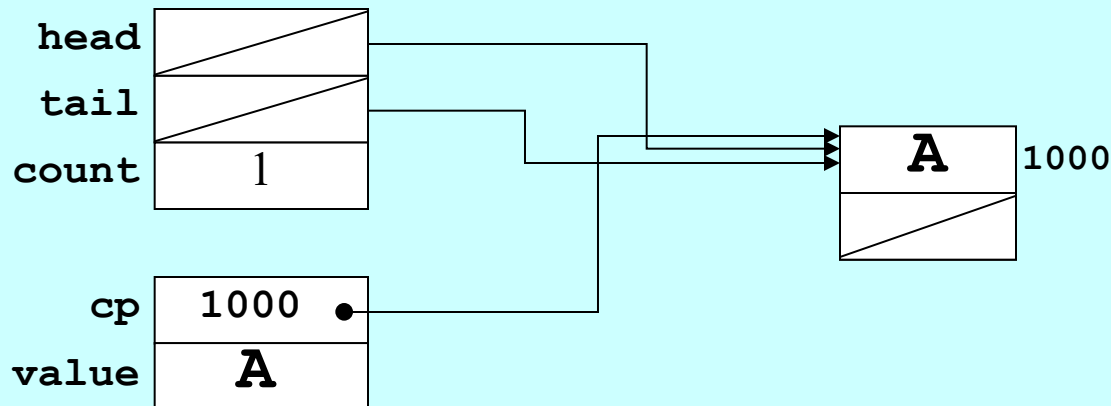
```
template <typename ValueType>  
Queue<ValueType>::Queue() {  
    head = tail = NULL;  
    count = 0;  
}
```



Tracing the List-Based Queue

```
Queue<char> queue;  
→ queue.enqueue('A')  
→ queue.enqueue('B')  
queue.dequeue()  
queue.enqueue('C')  
queue.dequeue()  
queue.dequeue()
```

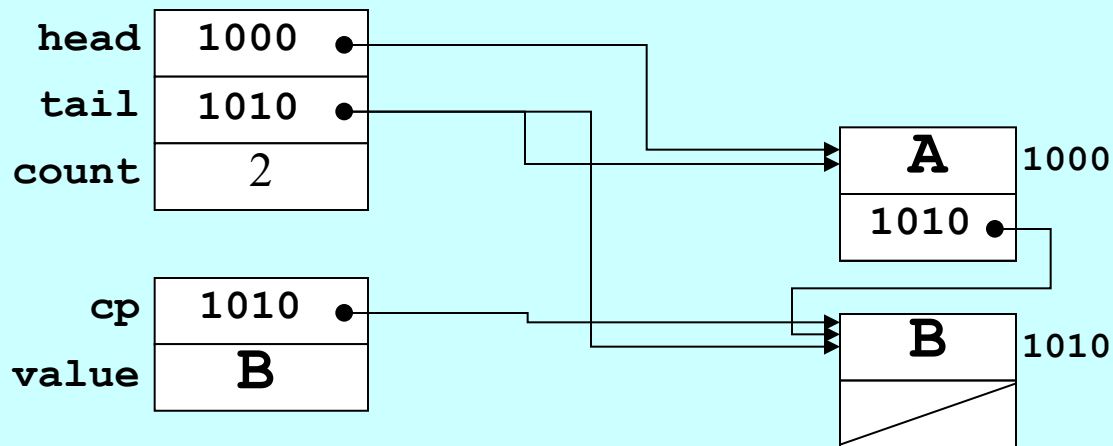
```
template ... enqueue(ValueType value) {  
    Cell *cp = new Cell;  
    cp->data = value;  
    cp->link = NULL;  
    if (head == NULL) {  
        head = cp;  
    } else {  
        tail->link = cp;  
    }  
    tail = cp;  
    count++;  
}
```



Tracing the List-Based Queue

```
Queue<char> queue;  
queue.enqueue('A')  
→ queue.enqueue('B')  
→ queue.dequeue()  
queue.enqueue('C')  
queue.dequeue()  
queue.dequeue()
```

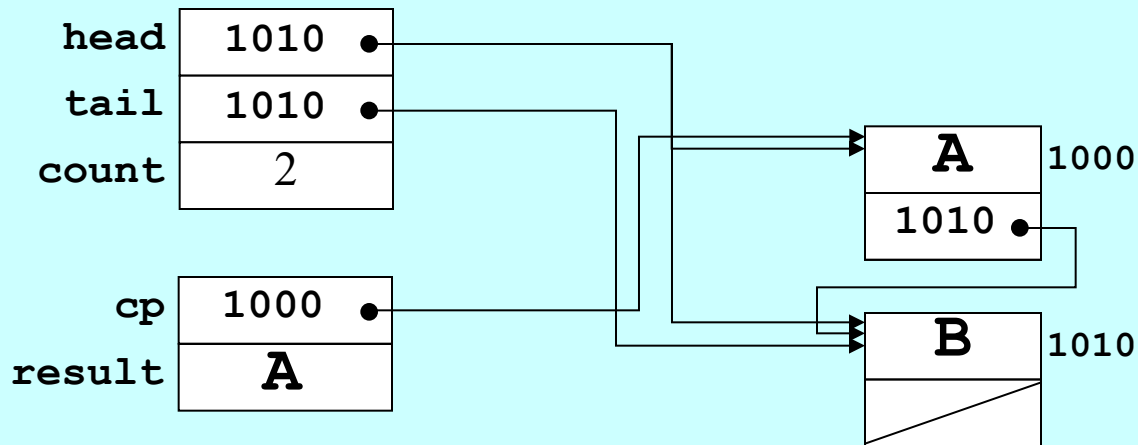
```
template ... enqueue(ValueType value) {  
    Cell *cp = new Cell;  
    cp->data = value;  
    cp->link = NULL;  
    if (head == NULL) {  
        head = cp;  
    } else {  
        tail->link = cp;  
    }  
    tail = cp;  
    count++;  
}
```



Tracing the List-Based Queue

```
Queue<char> queue;  
queue.enqueue('A')  
queue.enqueue('B')  
→ queue.dequeue() → 'A'  
→ queue.enqueue('C')  
queue.dequeue()  
queue.dequeue()
```

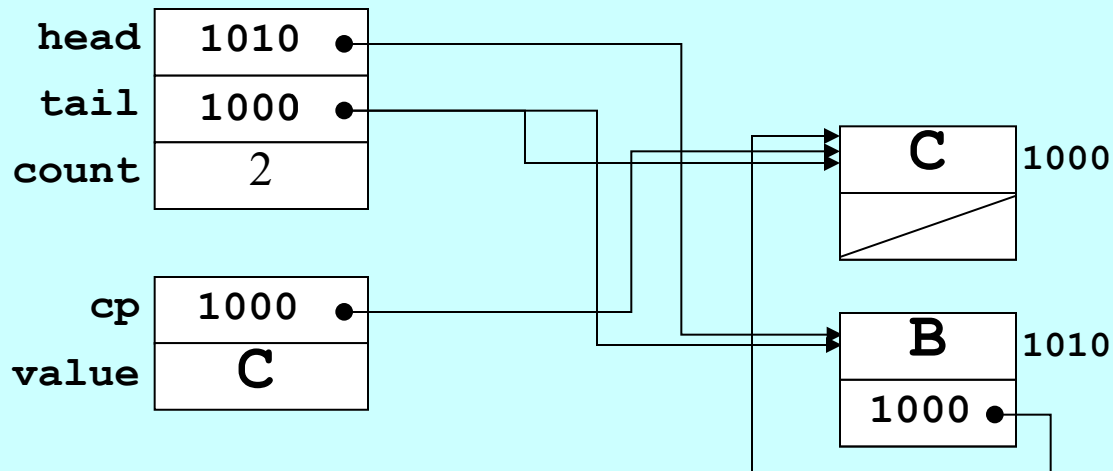
```
template <typename ValueType>  
ValueType Queue<ValueType>::dequeue() {  
    if (isEmpty()) error(...);  
    Cell *cp = head;  
    ValueType result = cp->data;  
    head = cp->link;  
    if (head == NULL) tail = NULL;  
    count--;  
    delete cp;  
    return result;  
}
```



Tracing the List-Based Queue

```
Queue<char> queue;  
queue.enqueue('A')  
queue.enqueue('B')  
queue.dequeue() → 'A'  
→ queue.enqueue('C')  
→ queue.dequeue()  
queue.dequeue()
```

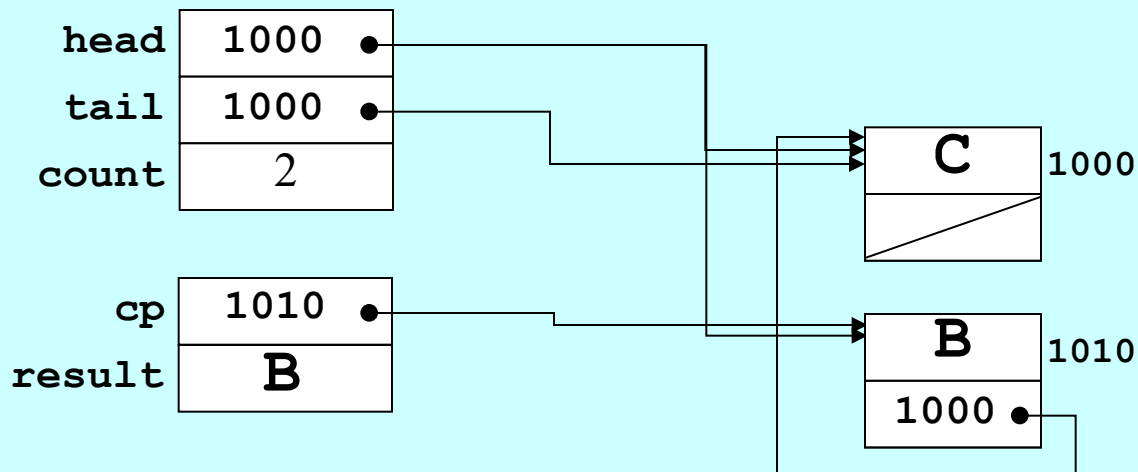
```
template ... enqueue(ValueType value) {  
    Cell *cp = new Cell;  
    cp->data = value;  
    cp->link = NULL;  
    if (head == NULL) {  
        head = cp;  
    } else {  
        tail->link = cp;  
    }  
    tail = cp;  
    count++;  
}
```



Tracing the List-Based Queue

```
Queue<char> queue;  
queue.enqueue('A')  
queue.enqueue('B')  
queue.dequeue() → 'A'  
queue.enqueue('C')  
→ queue.dequeue() → 'B'  
→ queue.dequeue()
```

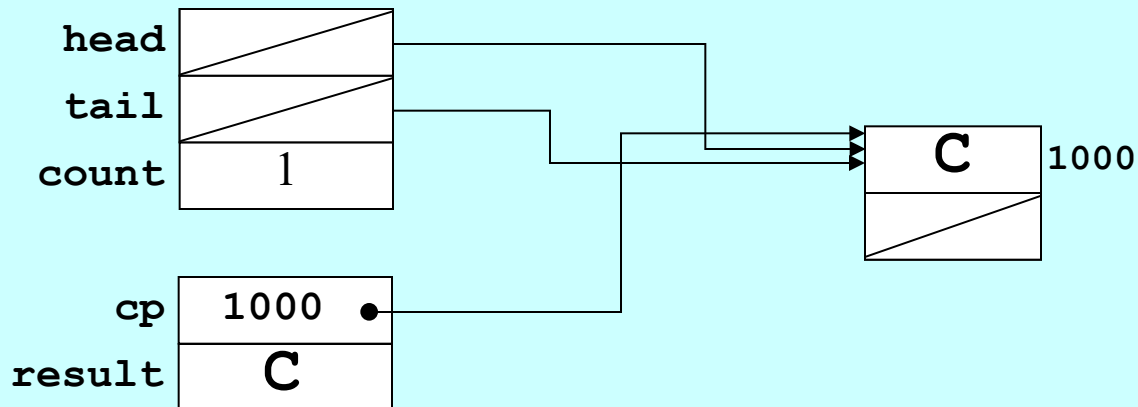
```
template <typename ValueType>  
ValueType Queue<ValueType>::dequeue() {  
    if (isEmpty()) error(...);  
    Cell *cp = head;  
    ValueType result = cp->data;  
    head = cp->link;  
    if (head == NULL) tail = NULL;  
    count--;  
    delete cp;  
    return result;  
}
```



Tracing the List-Based Queue

```
Queue<char> queue;  
queue.enqueue('A')  
queue.enqueue('B')  
queue.dequeue() → 'A'  
queue.enqueue('C')  
queue.dequeue() → 'B'  
→ queue.dequeue() → 'C'
```

```
template <typename ValueType>  
ValueType Queue<ValueType>::dequeue() {  
    if (isEmpty()) error(...);  
    Cell *cp = head;  
    ValueType result = cp->data;  
    head = cp->link;  
    if (head == NULL) tail = NULL;  
    count--;  
    delete cp;  
    return result;  
}
```



Code for the Linked-List Queue

```
/*
 * Implementation notes: Queue constructor
 * -----
 * The constructor must create an empty linked list and then
 * initialize the fields of the object.
 */

template <typename ValueType>
Queue<ValueType>::Queue() {
    head = tail = NULL;
    count = 0;
}

/*
 * Implementation notes: ~Queue destructor
 * -----
 * The destructor frees any memory that is allocated by the implementation.
 * Freeing this memory guarantees the client that the queue abstraction
 * will not "leak memory" in the process of running an application.
 * Because clear frees each element it processes, this implementation
 * of the destructor simply calls that method.
 */

template <typename ValueType>
Queue<ValueType>::~~Queue() {
    clear();
}
```

Code for the Linked-List Queue

```
/*
 * Implementation notes: size, isEmpty, clear
 * -----
 * The size and isEmpty field make use of the count field to avoid having
 * to count the elements. The clear method calls dequeue repeatedly to
 * ensure that the cells are freed.
 */

template <typename ValueType>
int Queue<ValueType>::size() {
    return count;
}

template <typename ValueType>
bool Queue<ValueType>::isEmpty() {
    return count == 0;
}

template <typename ValueType>
void Queue<ValueType>::clear() {
    while (count > 0) {
        dequeue();
    }
}
```

Code for the Linked-List Queue

```
/*
 * Implementation notes: enqueue
 * -----
 * This method allocates a new list cell and chains it in
 * at the tail of the queue.  If the queue is currently empty,
 * the new cell must also become the head pointer in the queue.
 */

template <typename ValueType>
void Queue<ValueType>::enqueue(ValueType value) {
    Cell *cp = new Cell;
    cp->data = value;
    cp->link = NULL;
    if (head == NULL) {
        head = cp;
    } else {
        tail->link = cp;
    }
    tail = cp;
    count++;
}
```



TUTORIAL

e for the Linked-List Queue

n notes: dequeue, peek

must check for an empty queue and report an
e is no first element. The dequeue method

* must also check for the case in which the queue becomes
* empty and set both the head and tail pointers to NULL.
*/

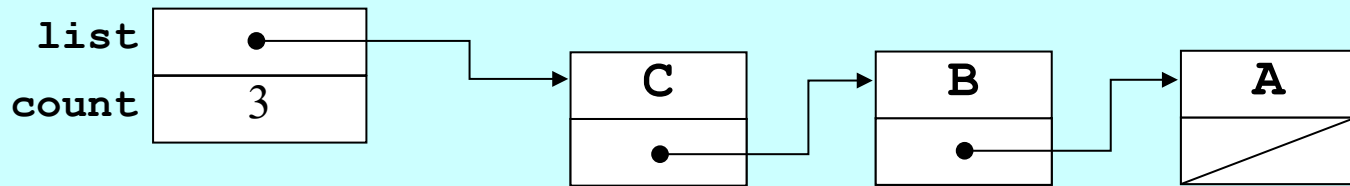
```
template <typename ValueType>
ValueType Queue<ValueType>::dequeue() {
    if (isEmpty()) error("dequeue: Attempting to dequeue an empty queue");
    Cell *cp = head;
    ValueType result = cp->data;
    head = cp->link;
    if (head == NULL) tail = NULL;
    count--;
    delete cp;
    return result;
}
```

```
template <typename ValueType>
ValueType Queue<ValueType>::peek() {
    if (isEmpty()) error("peek: Attempting to peek at an empty queue");
    return head->data;
}
```



Implementing Stacks Using Lists

- Now you see why the implementation of the *list-based stack* only needs to maintain a pointer at the top of the stack, and the linked list goes from the top to the bottom, because **pop** at the tail is difficult, similar to the **dequeue** at the tail case before.



- And you also see why the implementation of the *list-based stack* is much simpler than the corresponding code for the *list-based queue*.
- The differences between implementing a queue and a stack arise from the fact that the fundamental operations on a stack, **push** and **pop**, occur at the same end of the internal data structure, while in a queue, **enqueue** happens at one end, and **dequeue** happens at the other.

An Overly Simple Strategy

- The other straightforward way to represent the elements of a queue is to store the elements in an **array**, exactly as in the **Stack** class.
- Given this representation, the **enqueue** operation is extremely simple to implement. All you need to do is add the element to the end of the array and increment the element count. That operation runs in $O(1)$ time.
- The problem with this simplistic approach is that the **dequeue** operation requires removing the element from the beginning of the array. If you're relying on the same strategy you used for the **Stack** class, implementing this operation requires **moving all the remaining elements to fill the hole left by the dequeued element**. That operation therefore takes $O(N)$ time.

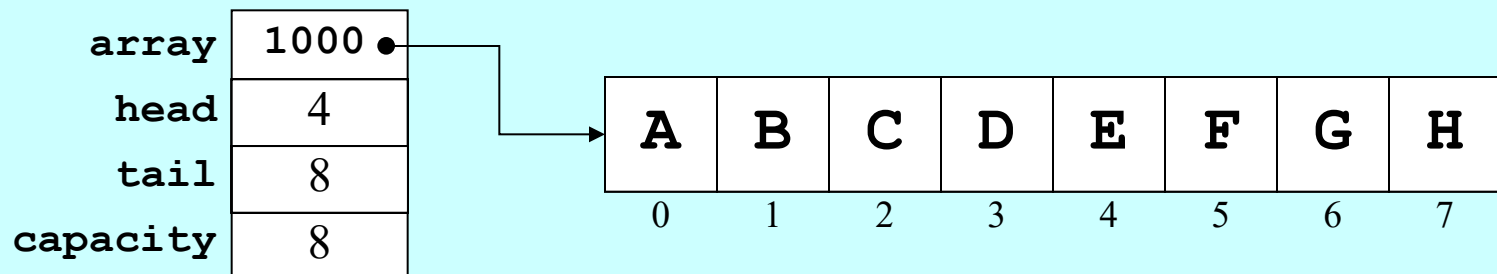
Fixing the $O(N)$ Problem

- The key to fixing the problem of having **dequeue** take $O(N)$ time is **to eliminate the need for any data motion by keeping track of two indices**: one to mark the head of the queue and another to mark the tail.
- Given these two indices, the **enqueue** operation stores the new element at the position marked by the **tail** index and then increments **tail** so that the next element is enqueued into the next slot. The **dequeue** operation is symmetric. The next value to be dequeued appears at the array position marked by the **head** index. Removing it is then simply a matter of incrementing **head**.
- Unfortunately, this strategy typically ends up filling the array space even when the queue itself contains very few elements, as illustrated on the next slide.

Tracing the Array-Based Queue

- Consider what happens if you execute the operations shown.
- Each **enqueue** operation adds a value at the tail of the queue.
- Each **dequeue** operation takes its result from the head.
- If you continue on in this way, what happens when you reach the end of the array space?

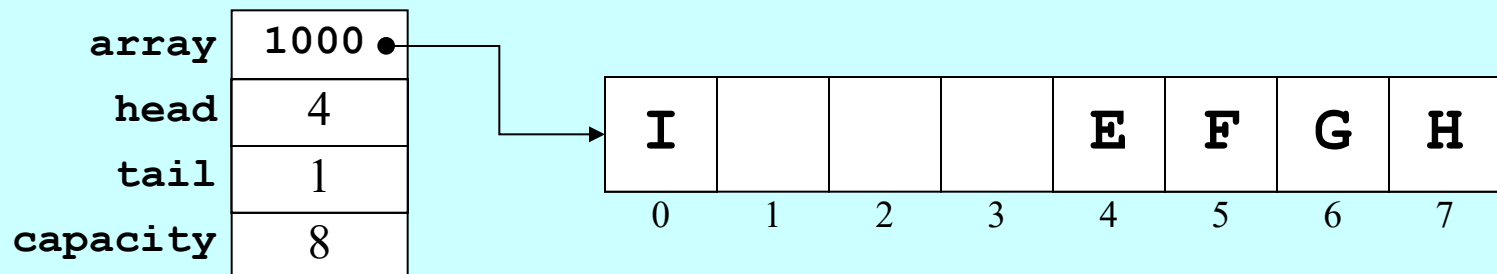
```
→ Queue<char> queue;  
→ queue.enqueue('A')  
→ queue.enqueue('B')  
→ queue.enqueue('C')  
→ queue.dequeue() → 'A'  
→ queue.enqueue('D')  
→ queue.enqueue('E')  
→ queue.dequeue() → 'B'  
→ queue.enqueue('F')  
→ queue.dequeue() → 'C'  
→ queue.dequeue() → 'D'  
→ queue.enqueue('G')  
→ queue.enqueue('H')  
→ queue.enqueue('I')
```



Tracing the Array-Based Queue

- At this point, enqueueing the **I** would require expanding the array, even though the queue contains only four elements.
- The solution to this problem is to let the elements cycle back to the beginning of the array.

```
Queue<char> queue;  
queue.enqueue('A')  
queue.enqueue('B')  
queue.enqueue('C')  
queue.dequeue() → 'A'  
queue.enqueue('D')  
queue.enqueue('E')  
queue.dequeue() → 'B'  
queue.enqueue('F')  
queue.dequeue() → 'C'  
queue.dequeue() → 'D'  
queue.enqueue('G')  
queue.enqueue('H')  
→ queue.enqueue('I')
```



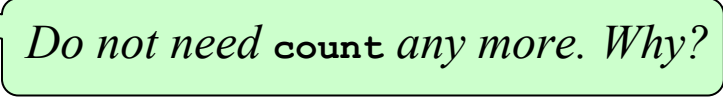


Implementing the Ring-Buffer Strategy

- The data structure described on the preceding slide is called a **ring buffer** because the end of the array is linked back to the beginning.
- The arithmetic operations necessary to implement the ring buffer strategy are easy to code using **modular arithmetic**, which is simply normal arithmetic in which all values are reduced to a specific range by dividing each result by some constant (in this case, the capacity of the array) and taking the **remainder**. In C++, you can use the % operator to implement modular arithmetic.
- When you are using the ring-buffer strategy, it is typically necessary to **expand the queue when there is still one free element left in the array**. If you don't do so, the simple test for an empty queue—whether **head** is equal to **tail**—fails because that would also be true in a completely full queue.

Structure for the Array-Based Queue

```
/* Private section */  
  
/*  
 * Implementation notes  
 * -----  
 * The array-based queue stores the elements in a ring buffer, which  
 * consists of a dynamic index and two indices: head and tail. Each  
 * index wraps to the beginning if necessary. This design requires  
 * that there is always one unused element in the array. If the queue  
 * were allowed to fill completely, the head and tail indices would  
 * have the same value, and the queue will appear empty.  
 */  
  
private:  
  
    static const int INITIAL_CAPACITY = 10;  
  
/* Instance variables */  
  
ValueType *array;           /* A dynamic array of the elements */  
int capacity;               /* The allocated size of the array */  
int head;                   /* The index of the head element */  
int tail;                   /* The index of the tail element */  
  
/* Private method prototypes */  
  
void expandCapacity();
```



Do not need count any more. Why?

Code for the Ring-Buffer Queue

```
/* Implementation section */

/*
 * Implementation notes: Queue constructor
 * -----
 * The constructor must allocate the array storage for the queue
 * elements and initialize the fields of the object.
 */

template <typename ValueType>
Queue<ValueType>::Queue() {
    capacity = INITIAL_CAPACITY;
    array = new ValueType[capacity];
    head = 0;
    tail = 0;
}

/*
 * Implementation notes: ~Queue destructor
 * -----
 * The destructor frees any memory that is allocated by the implementation.
 */

template <typename ValueType>
Queue<ValueType>::~~Queue() {
    delete[] array;
}
```

Code for the Ring-Buffer Queue

```
/*
 * Implementation notes: size
 * -----
 * The size of the queue can be calculated from the head and tail
 * indices by using modular arithmetic.
 */

template <typename ValueType>
int Queue<ValueType>::size() {
    return (tail + capacity - head) % capacity;
}

/*
 * Implementation notes: isEmpty
 * -----
 * The queue is empty whenever the head and tail indices are
 * equal. Note that this interpretation means that the queue
 * cannot be allowed to fill the capacity entirely and must
 * always leave one unused space.
 */

template <typename ValueType>
bool Queue<ValueType>::isEmpty() {
    return head == tail;
}
```

Code for the Ring-Buffer Queue

```
/*
 * Implementation notes: clear
 * -----
 * The clear method need not take account of where in the
 * ring buffer any existing data is stored and can simply
 * set the head and tail index back to the beginning.
 */

template <typename ValueType>
void Queue<ValueType>::clear() {
    head = tail = 0;
}

/*
 * Implementation notes: enqueue
 * -----
 * This method must first check to see whether there is
 * enough room for the element and expand the array storage
 * if necessary.
 */

template <typename ValueType>
void Queue<ValueType>::enqueue(ValueType value) {
    if (size() == capacity - 1) expandCapacity();
    array[tail] = value;
    tail = (tail + 1) % capacity;
}
```


Code for the Ring-Buffer Queue

```
/*
 * Implementation notes: dequeue, peek
 * -----
 * These methods must check for an empty queue and report an
 * error if there is no first element.
 */

template <typename ValueType>
ValueType Queue<ValueType>::dequeue() {
    if (isEmpty()) error("dequeue: Attempting to dequeue an empty queue");
    ValueType result = array[head];
    head = (head + 1) % capacity;
    return result;
}

template <typename ValueType>
ValueType Queue<ValueType>::peek() {
    if (isEmpty()) error("peek: Attempting to peek at an empty queue");
    return array[head];
}
```

Code for the Ring-Buffer Queue

```
/*
 * Implementation notes: expandCapacity
 * -----
 * This private method doubles the capacity of the elements array
 * whenever it runs out of space. To do so, it must allocate a new
 * array, copy all the elements from the old array to the new one,
 * and free the old storage. Note that this implementation also
 * shifts all the elements back to the beginning of the array.
 */

template <typename ValueType>
void Queue<ValueType>::expandCapacity() {
    int count = size();
    int oldCapacity = capacity;
    capacity *= 2;
    ValueType *oldArray = array;
    array = new ValueType[capacity];
    for (int i = 0; i < count; i++) {
        array[i] = oldArray[(head + i) % oldCapacity];
    }
    head = 0;
    tail = count;
    delete[] oldArray;
}

#endif
```

Implementing the **Vector** Class

- The **Vector** class resembles the **EditorBuffer** class in that it allows clients to insert and remove elements at any point in the sequence of values. At the same time, **Vector** resembles the **Stack** and **Queue** classes by implementing deep copying and by using templates to support polymorphism.
- The **Vector** class is typically implemented using a **dynamic array** as the underlying model. Despite the fact that the **Vector** class exports more methods, the code is similar to the array-based implementation of the **Stack** class.
- The only new feature of the **Vector** class is the definition of the *square-bracket* operators for selection, which requires providing a new definition of the **operator[]** method. Overloading this method makes it possible to use array-like selection notation on **Vector** objects.

The `vector.h` Interface

```
template <typename ValueType>
class Vector {
public:
    ...
    /*
    * Operator: []
    * Usage: vec[index]
    * -----
    * Overloads [] to select elements from this vector. This
    * extension enables the use of traditional array notation to
    * get or set individual elements. This method signals an error
    * if the index is outside the array range.
    */
    ValueType & operator[](int index);
    ...
private:
    ...
    ValueType *array;      /* Dynamic array of the elements */
    int capacity;          /* Allocated size of that array */
    int count;             /* Current count of elements in use*/
    ...
}
```

Redefining operator[]

```
/*
 * Implementation notes: Vector selection
 * -----
 * The following code implements traditional array selection using
 * square brackets for the index. The name of the method is
 * indicated by specifying the C++ keyword "operator" followed by
 * the operator symbol. To ensure that this operator returns an
 * assignable value, this method uses an & to return the result
 * by reference.
 */

template <typename ValueType>
ValueType & Vector<ValueType>::operator[](int index) {
    if (index < 0 || index >= count) {
        error("Vector selection index out of range");
    }
    return array[index];
}
```



Return by Reference

- The implementation of `operator[]` on the preceding slide uses a new feature of C++ called *return by reference*, which indicates that the return value is shared with the caller rather than copied. Remember `operator<<` and `operator=`?
- The syntax for return by reference is similar to that used for call by reference and involves adding an `&` token between the return type and the method name, as in the prototype

```
ValueType & Vector<ValueType>::operator[] (int index) ;
```

- Returning a value by reference means putting the result into an *lvalue*, which is an expression that can be used on the left side of an assignment. This interpretation makes it possible to assign new values to vector elements using an expression like

```
vec[i] = 17;
```



Cautionary Notes

- The return by reference feature of C++ is fraught with problems that can snare the unwary programmer.
- The most important caution is that you must **never use return by reference with a value that lives in the stack frame of the current method**. If you do, C++ will return the reference (or effectively the address) of that value to the caller, but the object to which that address refers has by that time disappeared, along with the stack frame in which it was created.
- As a general rule, it is probably best to **limit your use of return by reference** to situations that are essentially analogous to the selection brackets for vectors: cases in which you want to select a field from an abstract structure supplied by the caller; or when you want to return the same object that is passed into the current function using call by reference, e.g., assignment operator “=”, stream operation “<<”, etc.

Example: Return by Reference

```
#include <iostream>
using namespace std;

int & ReturnByReference(int a) // call by value, return by reference
{
    cout << "Into ReturnByReference:" << endl;
    cout << "&a = " << &a << "; a = " << a << endl;
    int* b = new int(a); // can be safely returned by reference
    cout << "&b = " << &b << "; b = " << b << "; *b = " << *b << endl;
    cout << "Out of ReturnByReference:" << endl;
    return *b;
}

int CallByReference(int & a) // call by reference, return by value
{
    cout << "Into CallByReference:" << endl;
    cout << "&a = " << &a << "; a = " << a << endl;
    cout << "Out of CallByReference:" << endl;
    return a;
}

int main()
{
    int a = 1;
    cout << "&a = " << &a << "; a = " << a << endl;
    a = CallByReference(ReturnByReference(a) = a+1);
    cout << "&a = " << &a << "; a = " << a << endl;
    return 0;
}
```

&a = 0x66FFE0; a = 1
Into ReturnByReference:
&a = 0x66FFA0; a = 1
&b = 0x66FFB0; b = 0x1000; *b = 1
Out of ReturnByReference:
Into CallByReference:
&a = 0x1000; a = 2
Out of CallByReference:
&a = 0x66FFE0; a = 2

Example: Return by Reference

```
#include <iostream>
using namespace std;

int & ReturnByReference(int a) // call by value, return by reference
{
    cout << "Into ReturnByReference:" << endl;
    cout << "&a = " << &a << "; a = " << a << endl;
    int* b = &a; // must not be returned by reference
    cout << "&b = " << &b << "; b = " << b << "; *b = " << *b << endl;
    cout << "Out of ReturnByReference:" << endl;
    return *b;
}

int CallByReference(int & a) // call by reference, return by value
{
    cout << "Into CallByReference:" << endl;
    cout << "&a = " << &a << "; a = " << a << endl;
    cout << "Out of CallByReference:" << endl;
    return a;
}

int main()
{
    int a = 1;
    cout << "&a = " << &a << "; a = " << a << endl;
    a = CallByReference(ReturnByReference(a) = a+1);
    cout << "&a = " << &a << "; a = " << a << endl;
    return 0;
}
```

&a = 0x66FFE0; a = 1
Into ReturnByReference:
&a = 0x66FFA0; a = 1
&b = 0x66FFB0; b = 0x66FFA0; *b = 1
Out of ReturnByReference:
Into CallByReference:
&a = 0x66FFA0; a = 7208656
Out of CallByReference:
&a = 0x66FFE0; a = 7208656

Integrating Prototypes and Code

- One of the primary reasons for using interfaces is to hide the complexity of the implementation from the eyes of the client. Everything the client needs to see goes into the .h file that defines the interface, relegating the messy details of the implementation to the corresponding .cpp file.
- C++ makes it difficult to maintain that level of separation as the interfaces become more sophisticated.
 - The **private section** of a class is included in the body of the class definition, which is in the .h file.
 - The entire implementation for **template** classes should be included in the interfaces.
- C++ even allows class definitions to include the bodies of the methods not only in the .h file, but directly within the classes.
- There are even syntactic advantages of adopting this strategy, e.g., the **template** keyword appears only once at the beginning of the class, you can eliminate the :: tags, etc.

Integrating Prototypes and Code

- However, the examples in the text still avoid combining code and prototypes to keep these aspects of the program as separate as possible.
- Moreover, there are other (advanced and complicated) ways to avoid combining code and prototypes, even for the template definitions, in the latest C++ standard updates, which are out of the scope of this course, and you don't need to worry about them for now.

The End