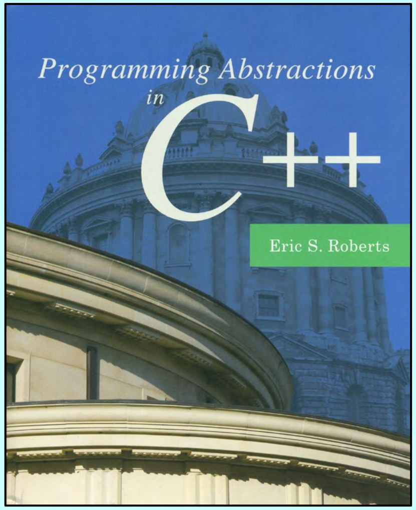
CHAPTER 12

Dynamic Memory Management



You have burdened your memory with exploded systems and useless names.

—Mary Shelley, *Frankenstein,* 1818

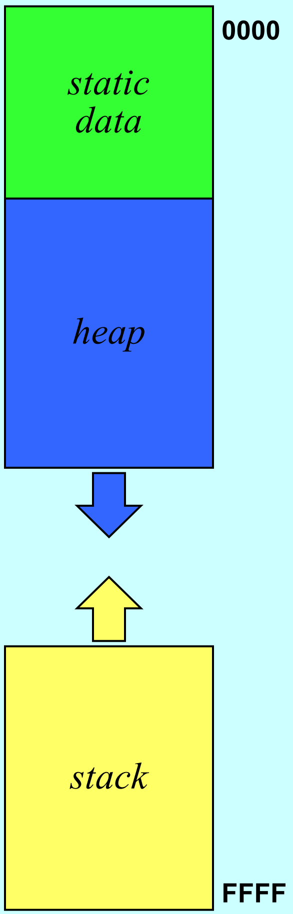
Pointers

- In C++, pointers serve several purposes, of which the following are the most important:
  - *Pointers allow you to refer to a large data structure in a compact way.* Because a memory address typically fits in a few bytes of memory, this strategy offers considerable space savings when the data structures themselves are large. E.g., ***call by pointers***.
  - *Pointers make it possible to reserve new memory during program execution*. In many applications, it is convenient to acquire new memory as the program runs and to refer to that memory using pointers, which is called ***dynamic allocation***.
  - *Pointers can be used to record relationships among data items*. Data structures that use pointers to create connections between individual components are called ***linked structures***. Programmers can indicate that one data item follows another in a conceptual sequence by including a pointer to the second item in the internal representation of the first.

Flashback

The Allocation of Memory to Variables

- When you declare a variable in a program, C++ allocates space for that variable from one of several memory regions.
- One region of memory is reserved for program code and global variables/constants that persist throughout the lifetime of the program. This information is called ***static data***.
- Each time you call a method, C++ allocates a new block of memory called a ***stack frame*** to hold its local variables. These stack frames come from a region of memory called the ***stack***.
- It is also possible to allocate memory dynamically, as we will describe in Chapter 12. This space comes from a pool of memory called the ***heap***.
- In classical architectures, the stack and heap grow toward each other to *maximize the available space*.



Dynamic Allocation

- C++ uses the **`new`** operator to allocate memory on the heap.
- You can allocate a single value (as opposed to an array) by writing **`new`** followed by the type name. Thus, to allocate space for a **`int`** on the heap, you would write:

```
int * pi = new int;
```

- You can allocate an array of values using the following form:

```
type * name = new type[size];
```

Thus, to allocate an array of 10000 integers, you would write:

```
int * arr = new int[10000];
```

- The **`delete`** operator frees memory previously allocated. For arrays, you need to include empty brackets, as in:

```
delete pi;
delete [] arr;
```

How does the compiler know how much memory to release?

Using `delete`

- Always pair a **`delete`** to a **`new`**, and a **`delete []`** to a **`new []`**.
- No *size* needs to be specified for **`delete []`**. One of the approaches for compilers know how much memory to free is to allocate a little more memory and to store a count of the elements in a head segment (invisible to you) just before the first array element.
- **`delete ptr`** only frees the memory space pointed by **`ptr`**, but not the memory space occupied by **`ptr`** itself. **`ptr`** is still a live variable until it is released (depending on how it is declared), though it is ***dangling*** (i.e., does not point to a valid object of the appropriate type).
- To avoid dangling pointers, after deleting a pointer (freeing the memory space it points to), one can ***nullify*** a pointer by:

```
ptr = NULL; // nullptr since C++11
```

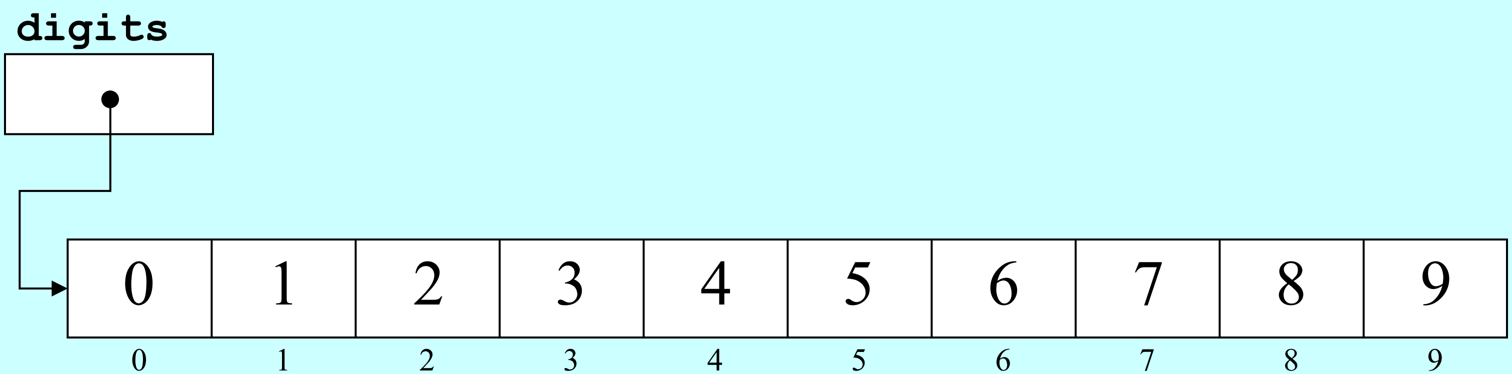
  But there might be other pointers pointing to the same address that are not nullified.

Exercise: Dynamic Arrays

- Write a method **`createIndexArray(n)`** that returns an integer array of size **`n`** in which each element is initialized to its index. As an example, calling

```
Int * digits = createIndexArray(10);
```

  should result in the following configuration:



- How would you free the memory allocated by this call?

Exercise: Dynamic Arrays

```
/*
 * Function: createIndexArray
 * Usage: int *array = createIndexArray(n);
 * ----------------------------------------
 * Returns a new dynamic integer array of length n
 * in which every element is set to its own index.
 */

int* createIndexArray(int n) {
   int* array = new int[n];
   for (int i = 0; i < n; i++) {
      array[i] = i;
   }
   return array;
}

int main() {
   int *digits = createIndexArray(10);
   delete [] digits;
}
```

If we change dynamic array to automatic:
`int array[n];`
what happens?

If we change `10` *to* `1`*, do we still need* `[]` *after* `delete`*?*
Yes.

Garbage collection

- The big challenge in working with dynamic memory allocation is freeing the heap memory you allocate. Programs that fail to do so have what computer scientists call ***memory leaks***.
- In Python and Java, objects are created on the heap and are automatically reclaimed by a ***garbage collector*** when those objects are no longer accessible. This discipline makes memory management very easy and convenient (for you).
- Garbage collection frees the programmer from manually dealing with memory deallocation. As a result, certain categories of bugs are eliminated or substantially reduced:
  - Certain kinds of memory leaks;
  - Dangling pointer bugs;
  - Double free bugs;
  - Efficient implementations of persistent data structures, etc.

```
char *p1 = new char;
char *p2 = new char;
p2 = p1;
delete p1;
p1 = NULL;
delete p2;
```

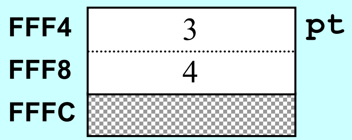
C++ Memory Management

- Garbage collection, however, is not always efficient:
  - Consuming additional resources;
  - Performance impacts;
  - Possible stalls in program execution;
  - Incompatibility with manual resource management, etc.
- For performance, garbage collection is not adopted in C++:
  - Objects can be allocated either on the stack or in the heap.
  - Programmers must manage heap memory allocation explicitly.
- In object-oriented programming, it is nearly impossible for clients to remember exactly what heap storage is currently active when using objects. Fortunately, the designers of C++ made it possible to free heap memory allocated by objects (clients) when those objects ***go out of scope*** on the stack.
- In well-designed C++ programs, (the implementer of) each class takes responsibility for its own heap storage.

Allocating a `Point` Object

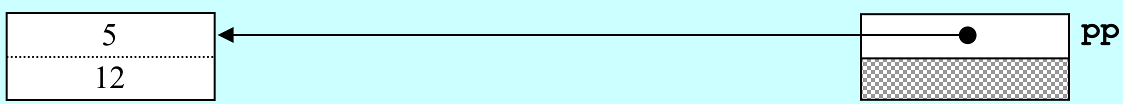
- The usual way to allocate a **`Point`** object is to declare it as a local variable on the stack, as follows:

```
Point pt(3, 4);
```



- It is, however, also possible to allocate a **`Point`** object on the heap using the following code:

```
Point* pp = new Point(5, 12);
```



- However, because of the data protection mechanism, it is nearly impossible for clients to know exactly what heap storage is currently active when deleting objects. For instance, some of the data fields might have been using the heap memory dynamically allocated by the member methods.

Destructors

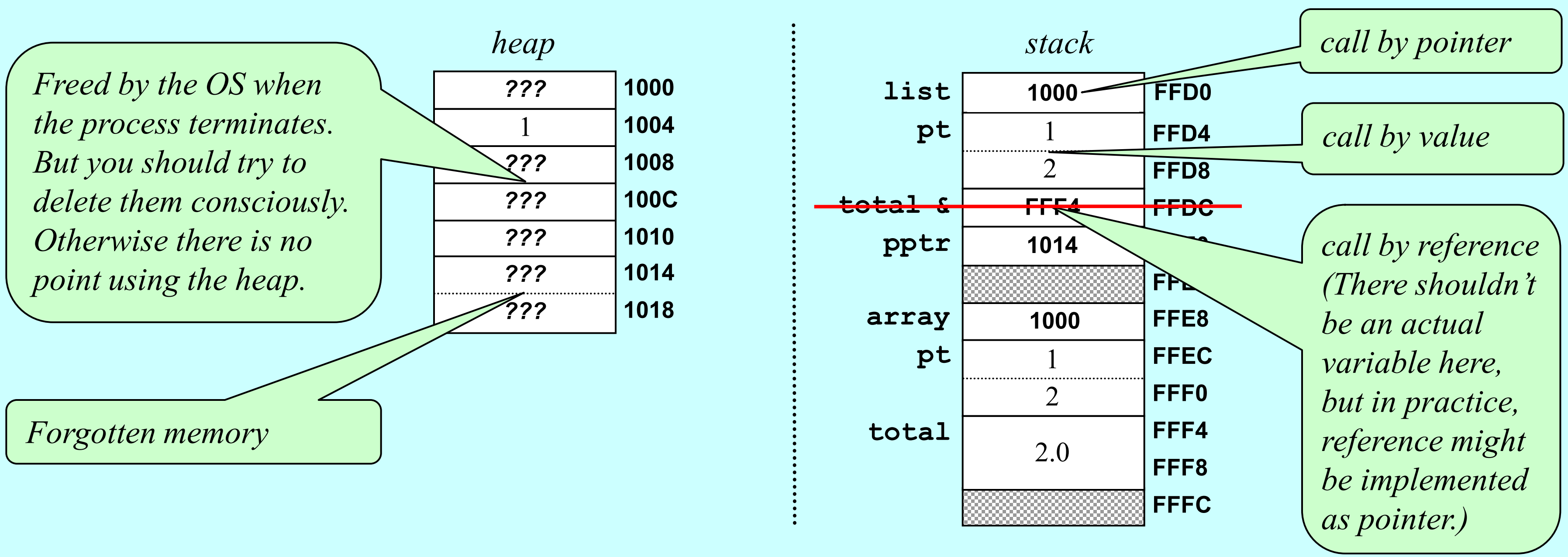
- In C++, class definitions often include a ***destructor***, which specifies how to free the storage used to represent an instance of that class.
- The prototype for a destructor has no return type and uses the name of the class preceded by a tilde (~). The destructor must not take any arguments.
- C++ calls the destructor automatically whenever a variable of a particular class is released. For stack objects, this happens when the function returns. The effect of this rule is that a C++ program that declares its objects as local variables on the stack will automatically reclaim those variables.
- If you instead allocate space for an object in the heap using `new`, you must explicitly free that object by calling `delete`. Calling `delete` also automatically invokes the destructor.
- If `new` is used in constructors, `delete` should most probably be used in destructors too.

Heap-Stack Diagrams

- It is easier to understand how C++ works if you have a good mental model of its use of memory. One of the most useful models is a ***heap-stack diagram***, which shows the heap on the left and the stack on the right, separated by a dotted line.
- Whenever your program uses **`new`**, you need to add a block of memory to the heap side of the diagram. That block must be large enough to store the entire value you're allocating. If the value is a **`struct`** or an object type, that block must include space for all the members inside that structure.
- Whenever your program calls a method, you need to create a new stack frame by adding a block of memory to the stack side. For method calls, you need to add enough space to store the local variables for the method, again with some overhead information that tracks what the program is doing. When a method returns, C++ reclaims the memory in its frame.

Exercise: Heap-Stack Diagrams

```
int main() {
void nonsense(int list[], Point pt, double & total) {
    Point *pptr = new Point;
    list[1] = pt.x;
    total += pt.y;
}
```



Dynamic Allocation in C

- Supported in `<cstdlib>` (`stdlib.h`)

Dynamic memory management

calloc	Allocate and zero-initialize array (function)
free	Deallocate memory block (function)
malloc	Allocate memory block (function)
realloc	Reallocate memory block (function)

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

Dynamic Allocation in Modern C++

- Supported in `<memory>` starting from C++11

Managed pointers

auto_ptr	Automatic Pointer [deprecated] (class template)
auto_ptr_ref	Reference to automatic pointer (class template)
shared_ptr	Shared pointer (class template)
weak_ptr	Weak shared pointer (class template)
unique_ptr	Unique pointer (class template)
default_delete	Default deleter (class template)

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

Dynamic Allocation in Modern C++

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

int main() {
  // Create a unique_ptr, <memory>, C++11
  unique_ptr<int> p1(new int(1));
  // Create a unique_ptr using make_unique, <memory>, C++14
  unique_ptr<int> p2 = make_unique<int>(2);
  // Transfer ownership from p1 to p3, <utility>, C++11
  unique_ptr<int> p3 = move(p1);

  if (!p1) {
    cout << "p1 is now null" << endl;
  }
  cout << *p2 << *p3 << endl;

  return 0;
}
```

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

Dynamic Allocation in Modern C++

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

int main() {
  // Create a shared_ptr, <memory>, C++11
  shared_ptr<int> ptr1 = make_shared<int>(10);
  // Make a new shared_ptr to share ownership with ptr1
  shared_ptr<int> ptr2 = ptr1;
  cout << ptr1.use_count() << endl;
  // Decreases the reference count by 1 using reset
  ptr1.reset();
  cout << ptr2.use_count() << endl;
  // Memory is not deleted yet because ptr2 still owns it
  cout << *ptr2 << std::endl;

  return 0;
}
```

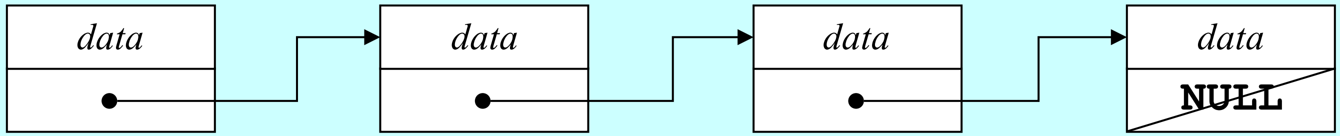
Things that you only need to know that you don't know. I don't need you to memorize it for the exam.

Pointers

- In C++, pointers serve several purposes, of which the following are the most important:
  - *Pointers allow you to refer to a large data structure in a compact way.* Because a memory address typically fits in a few bytes of memory, this strategy offers considerable space savings when the data structures themselves are large. E.g., ***call by pointers***.
  - *Pointers make it possible to reserve new memory during program execution*. In many applications, it is convenient to acquire new memory as the program runs and to refer to that memory using pointers, which is called ***dynamic allocation***.
  - *Pointers can be used to record relationships among data items*. Data structures that use pointers to create connections between individual components are called ***linked structures***. Programmers can indicate that one data item follows another in a conceptual sequence by including a pointer to the second item in the internal representation of the first.

Linking Objects Together

- Pointers are important in programming because they make it possible to represent the relationship among objects by linking them together in various ways.
- The simplest example of a linked structure (which appears first in this chapter and is used throughout the later chapters) is called a ***linked list***, in which each object in a sequence contains a reference to the one that follows it:



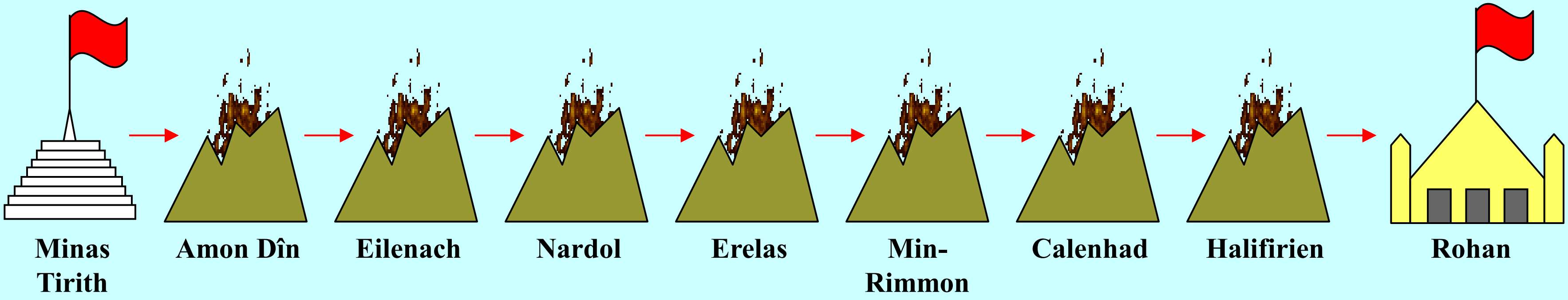
- C++ marks the end of linked list using the constant **`NULL`**, which signifies a pointer that does not have an actual target.
- In diagrams, the **`NULL`** value marking the end of a list is often indicated by drawing a diagonal line across the box.

The Beacons of Gondor

For answer Gandalf cried aloud to his horse. "On, Shadowfax! We must hasten. Time is short. See! The beacons of Gondor are alight, calling for aid. War is kindled. See, there is the fire on Amon Dîn, and flame on Eilenach; and there they go speeding west: Nardol, Erelas, Min-Rimmon, Calenhad, and the Halifirien on the borders of Rohan."

—J. R. R. Tolkien, *The Return of the King,* 1955

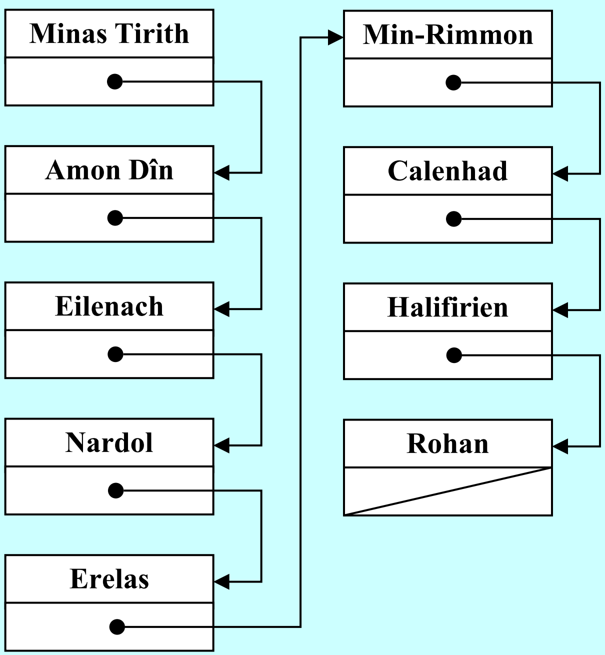
In a scene that was brilliantly captured in Peter Jackson's film adaptation of *The Return of the King,* Rohan is alerted to the danger to Gondor by a succession of signal fires moving from mountain top to mountain top. This scene is a perfect illustration of the idea of message passing in a linked list.



Message Passing in Linked Structures

To represent this message-passing image, you might use a definition such as the one shown on the right.

You can then initialize a chain of **`Tower`** structures, like this:



Calling **`signal`** on the first tower sends a message down the chain.

```
struct Tower {
   string name;   /* The name of this tower    */
   Tower *link;   /* Pointer to the next tower */
};

/*
 * Function: createTower(name, link);
 * ----------------------------------
 * Creates a new Tower with the specified values.
 */

Tower *createTower(string name, Tower *link) {
   Tower *tp = new Tower;
   tp->name = name;
   tp->link = link;
   return tp;
}

/*
 * Function: signal(start);
 * ------------------------
 * Generates a signal beginning at start.
 */

void signal(Tower *start) {
   if (start != NULL) {
      cout << "Lighting " << start->name << endl;
      signal(start->link);
   }
}
```

Does the new tower go to the head or tail of the linked list of old towers?

Example: The `CharStack` Class

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

`CharStack cstk;` Initializes an empty stack.
`cstk.size()` Returns the number of characters pushed onto the stack.
`cstk.isEmpty()` Returns **`true`** if the stack is empty.
`cstk.clear()` Deletes all characters from the stack.
`cstk.push(ch)` Pushes a new character onto the stack.
`cstk.pop()` Removes and returns the top character from the stack.

The `charstack.h` Interface

```
/*
 * File: charstack.h
 * -----------------
 * This interface defines the CharStack class.
 */

#ifndef _charstack_h
#define _charstack_h

class CharStack {
public:

/*
 * CharStack constructor and destructor
 * ------------------------------------
 * The constructor initializes an empty stack.  The destructor
 * is responsible for freeing heap storage.
 */

   CharStack();
   ~CharStack();
```

The `charstack.h` Interface

```
/*
 * Methods: size, isEmpty, clear, push, pop
 * ----------------------------------------
 * These methods work exactly as they do for the Stack class.
 * The peek method is deleted here to save space.
 */

   int size();
   bool isEmpty();
   void clear();
   void push(char ch);
   char pop();

#include "charstackpriv.h"

}

#endif
```

The **charstackpriv.h** File

```
/*
 * File: charstackpriv.h
 * ---------------------
 * This file contains the private data for the CharStack class.
 */

private:

/* Instance variables */

   char *array;            /* Dynamic array of characters   */
   int capacity;           /* Allocated size of that array  */
   int count;              /* Current count of chars pushed */

/* Private function prototypes */

   void expandCapacity();
```

The **charstack.cpp** Implementation

```
/*
 * File: charstack.cpp
 * -------------------
 * This file implements the CharStack class.
 */

#include "charstack.h"
#include "error.h"
using namespace std;

/*
 * Constant: INITIAL_CAPACITY
 * --------------------------
 * This constant defines the initial allocated size of the dynamic
 * array used to hold the elements.  If the stack grows beyond its
 * capacity, the implementation doubles the allocated size.
 */

const int INITIAL_CAPACITY = 10;
```

The `charstack.cpp` Implementation

```
/*
 * Implementation notes: constructor and destructor
 * ------------------------------------------------
 * The constructor allocates dynamic array storage to hold the
 * stack elements.  The destructor must free these elements
 */

CharStack::CharStack() {
   capacity = INITIAL_CAPACITY;
   array = new char[capacity];
   count = 0;
}

CharStack::~CharStack() {
   delete[] array;
}
```

The `charstack.cpp` Implementation

```
int CharStack::size() {
   return count;
}

bool CharStack::isEmpty() {
   return count == 0;
}

void CharStack::clear() {
   count = 0;
}

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

char CharStack::pop() {
   if (isEmpty()) error("pop: Attempting to pop an empty stack");
   return array[--count];
}
```

The `charstack.cpp` Implementation

```
/*
 * Implementation notes: expandCapacity
 * ------------------------------------
 * This private method doubles the capacity of the elements array
 * whenever it runs out of space.  To do so, it must copy the
 * pointer to the old array, allocate a new array with twice the
 * capacity, copy the characters from the old array to the new
 * one, and finally free the old storage.
 */

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;
}
```

Copying Objects

- There is one remaining issue, from previous chapters, about creating new abstract classes that is extremely important in practice, which is how such objects behave if you copy them.
- When you are defining a new abstract data type in C++, you typically need to define two methods to ensure that copies are handled correctly:
  - The operator **`operator=`**, which takes care of assignments
  - A copy constructor, which takes care of by-value parameters
- These methods have well-defined signatures and structures, and the easiest thing to do is simply to copy the sample code on the next few slides, adapting it as necessary to account for the specific instance variables that need to be copied in the underlying representation.

Copying Objects

- The process of copying an object is controlled by two standard methods, each of which has a specific prototype.
- The ***assignment operator*** has the following form:

Return by reference

```
type & type::operator=(const type & src)
```

Call by reference

- The prototype for the ***copy constructor*** looks like this:

Constructors have the same name as the class (type).

```
type::type(const type & src)
```

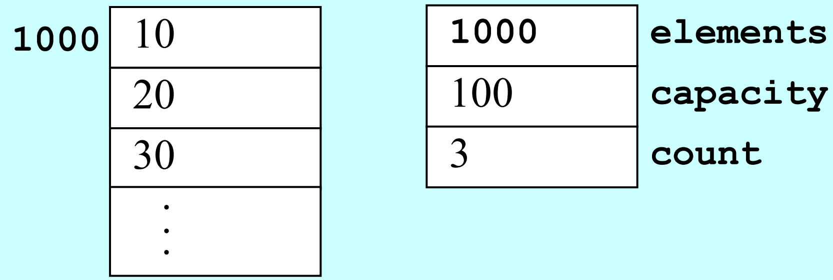
- Call/return by value might cause unnecessary calls to the copy constructor. Constant call by reference protects the source.
- An assignment operator can return anything it wants, but the standard C and C++ assignment operators return a reference to the left-hand operand (e.g., for primitive types). This allows you to chain assignments together. Unless you have a really good reason, you want to follow this convention.

Assignment and Copy Constructors

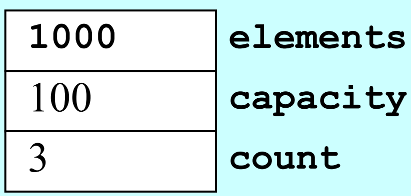
- If you don't provide assignment operator and copy constructor yourself, C++ provides them for you.
- Unfortunately, the default behavior of C++ is to copy only the top-level fields in an object, which means that all dynamically allocated memory is shared between the original and the copy.
- This default behavior, which is called ***shallow copying***, violates the semantics of data structures such as the collection classes. The collection classes in C++ are defined so that copying one collection to another creates an entirely new copy of the collection. What you need to implement instead is ***deep copying***, which copies the data in the dynamically allocated memory as well.
- The simplest strategy for ensuring that clients don’t violate the integrity of data structures through assignment is to prevent the client from copying an object altogether by making the assignment operator and copy constructor private methods. (Remember streams?)

Shallow *vs.* Deep Copying

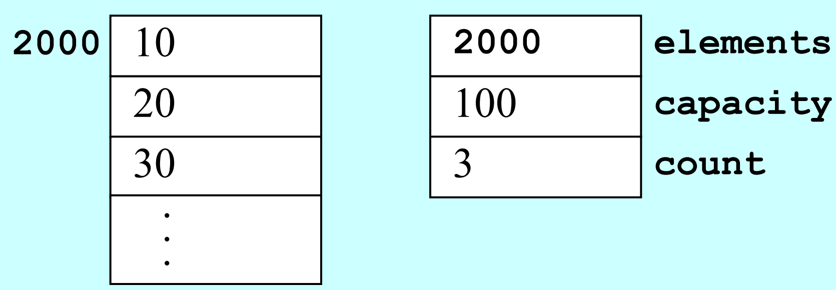
- Suppose that you have an existing `Vector<int>` with three elements as shown in the diagram to the right.



- A ***shallow copy*** allocates new fields for the object itself and copies the information from the original. Unfortunately, the dynamic array is copied as an address, not the data.



- A ***deep copy*** also copies the contents of the dynamic array and therefore creates two independent structures



Implementing Deep Copy Semantics

- This becomes an issue especially when you implement a class that uses dynamic memory.
- The crux of the problem is that copying an abstract data object typically needs to copy the underlying data (***deep copying***) and not just the fields directly accessible in the object.
- Unfortunately, the default interpretation in C++ is to copy only the top-level fields (***shallow copying***), which can lead to serious errors if you are expecting deep copying.
- Therefore, it is important to specify how the object can be properly copied in your implementation.

Code to Implement Deep Copying

```
/*
 * Implementation notes: copy constructor and assignment operator
 * --------------------------------------------------------------
 * These methods make it possible to pass a CharStack by value or
 * assign one CharStack to another.
 */

CharStack::CharStack(const CharStack & src) {
   deepCopy(src);
}

CharStack & CharStack::operator=(const CharStack & src) {
   if (this != &src) {
      delete[] array;
      deepCopy(src);
   }
   return *this;
}

void CharStack::deepCopy(const CharStack & src) {
   array = new char[src.count]; capacity (a typo in the textbook, at least in my copy)
   for (int i = 0; i < src.count; i++) {
      array[i] = src.array[i];
   }
   count = src.count;
   capacity = src.capacity;
}
```

Flashback

Unit Testing

- One of the most important responsibilities you have as a programmer is to test your code as thoroughly as you can. Although testing can never guarantee the absence of errors, adopting a deliberate testing methodology helps you to find at least some of the errors in your code.
- Whenever you write a module, it is good practice to create a test program that checks the correctness of that module in isolation from the rest of the code. Such tests are called ***unit tests***.
- The text uses the **`assert`** macro from the **`<cassert>`** library to implement the unit-testing strategy. If the conditional expression in the **`assert`** macro is **`true`**, execution continues normally. If the expression is **`false`**, the **`assert`** macro prints a message identifying the source of the error and exits from the program.

FIGURE 12-10 Unit test for the CharStack class

```
/*
 * File: CharStackUnitTest.cpp
 * ---------------------------
 * This file contains a unit test of the CharStack class that uses the
 * C++ assert macro to test that each operation performs as it should.
 */

#include <iostream>
#include <cassert>
#include "charstack.h"
using namespace std;

int main() {
   CharStack cstk;                              /* Declare an empty CharStack  */
   assert(cstk.size() == 0);                    /* Make sure its size is 0     */
   assert(cstk.isEmpty());                      /* And that isEmpty is true    */
   cstk.push('A');                              /* Push the character 'A'      */
   assert(!cstk.isEmpty());                     /* The stack is now not empty  */
   assert(cstk.size() == 1);                    /* And has size 1              */
   assert(cstk.peek() == 'A');                  /* Check that peek returns 'A' */
   cstk.push('B');                              /* Push the character 'B'      */
   assert(cstk.peek() == 'B');                  /* Make sure peek returns it   */
   assert(cstk.size() == 2);                    /* And that the size is now 2  */
   assert(cstk.pop() == 'B');                   /* Pop and test for the 'B'    */
   assert(cstk.size() == 1);                    /* Recheck the size            */
   assert(cstk.peek() == 'A');                  /* And make sure 'A' is on top */
   cstk.push('C');                              /* Test a push after a pop     */
   assert(cstk.size() == 2);                    /* Make sure size is correct   */
   assert(cstk.pop() == 'C');                   /* And that pop returns a 'C'  */
   assert(cstk.peek() == 'A');                  /* The 'A' is now back on top  */
   assert(cstk.pop() == 'A');                   /* Pop and test for the 'A'    */
   assert(cstk.size() == 0);                    /* And make sure size is 0     */
   for (char ch = 'A'; ch <= 'Z'; ch++) {       /* Push the entire alphabet    */
      cstk.push(ch);                            /*   one character at a time   */
   }                                            /*   to test stack expansion   */
   assert(cstk.size() == 26);                   /* Make sure the size is 26    */
   for (char ch = 'Z'; ch >= 'A'; ch--) {       /* Pop the characters in       */
      assert(cstk.pop() == ch);                 /*   reverse order to make     */
   }                                            /*   sure they're all there    */
   assert(cstk.isEmpty());                      /* Ensure the stack is empty   */
   for (char ch = 'A'; ch <= 'Z'; ch++) {       /* Push the alphabet again to  */
      cstk.push(ch);                            /*   test that it works after  */
   }                                            /*   expansion                 */
   assert(cstk.size() == 26);                   /* Check size is again 26      */
   cstk.clear();                                /* Check the clear method      */
   assert(cstk.size() == 0);                    /* And check if stack is empty */
   cstk.clear();                                /* Test clear with empty stack */
   assert(cstk.size() == 0);
   cout << "CharStack unit test succeeded" << endl;
   return 0;
}
```

The Uses of `const`

- The **`const`** keyword has many distinct purposes in C++. This text uses it in the following three contexts:

  1. ***Constant definitions***. Adding the keyword **`const`** to a variable definition tells the compiler to disallow subsequent assignments to that variable, thereby making it constant.
     ```
     const double PI = 3.14159265358979323846;
     static const int INITIAL_CAPACITY = 10;
     ```
  2. ***Constant call by reference***. Adding **`const`** to the declaration of a reference parameter signifies that the function will not change the value of that parameter. This guarantee allows the compiler to share the contents of a data structure without allowing methods to change it.
     ```
     void deepCopy(const CharStack & src);
     ```
  3. ***Constant methods***. Adding **`const`** after the parameter list of a method guarantees that the method will not change the object.
     ```
     int CharStack::size() const
     ```

- Classes that use the **`const`** specification for all appropriate parameters and methods are said to be **`const`**-correct.

Class Constants

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

class A {
public:
   const int c1;
   const int c2;
   A();
   A(int x, int y);
};

A::A():c1(1),c2(2){}
A::A(int x, int y):c1(x),c2(y){}

int main()
{
   A a, b(3, 4);
   cout << a.c1 << a.c2 << endl;
   cout << b.c1 << b.c2 << endl;
}
```

You can only use the initializer list to initialize a class constant.

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

class A {
public:
   const int c1 = 1; // Since C++11
   const int c2 = 2; // Since C++11
};

int main()
{
   A a, b;
   cout << a.c1 << a.c2 << endl;
   cout << b.c1 << b.c2 << endl;
   cout << (&a.c1 == &b.c1) << endl;
   cout << (&a.c2 == &b.c2) << endl;
}
```

Compile-time constants that do not require storage allocation

They have different addresses even though they share the same constant value, and you can even re-initialize them using initializer list in the constructor functions.

Static Class Members

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

class A {
public:
   static const int c = 1;
   static int i;
};

int A::i = 2;

int main()
{
   A a, b;
   cout << a.c << a.i << endl;
   cout << b.c << b.i << endl;
   cout << (&a.c == &b.c) << endl;
   cout << (&a.i == &b.i) << endl;
   a.i = 3;
   cout << b.i << endl;
}
```

Static member variables must be defined outside the class definition because they are not tied to any specific instance of the class but rather to the class itself. This definition allocates storage for the variable and sets its initial value.

Now they will have the same address and the same value.

Flashback

The `random.cpp` Implementation

```
/*
 * Implementation notes: initRandomSeed
 * ------------------------------------
 * The initRandomSeed function declares a static variable that
 * keeps track of whether the seed has been initialized.  The
 * first time initRandomSeed is called, initialized is false,
 * so the seed is set to the current time.
 */

void initRandomSeed() {
   static bool initialized = false;
   if (!ini   alized) {
      srand(i     me(NULL)));
      initializ      e;
   }
}
```

The lifetime of **`static`** *variables begins the first time the program flow encounters the declaration and it ends at program termination. The compiler allocates only one copy of* **`initialized`***, which is initialized exactly once, and then shared by all calls to* **`initRandomSeed`***. This ensures that the initialization step must be performed once and only once.*

The Uses of `static`

Applied to	Meaning
a local variable	The variable is "permanent", in the sense that it is initialized only once and retains its value from one function call to the next. It is like having a global variable with local scope.
a global constant	Since a global constant has internal linkage by [illegible] variables, which have external linkage by defa[illegible] available for use in the file in which it is defi[illegible] effect, static const).
a global variable or a free function	The scope of the function, or the variable, is limited to the file in which it is defined. Without a static qualifier, any free function or global variable in a file has the extern qualifier by default, making it visible from other files in the compilation unit.
a member variable of a class.	There is only one such variable for the class, no matter how many objects of the class are created. In other words, it turns the member variable from an "instance variable" into a "class variable".
a member function of a class.	The function may access only static members of the class. That is, it may not access any instance members (since there may not be any, if no objects of the class have been created).

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

The End