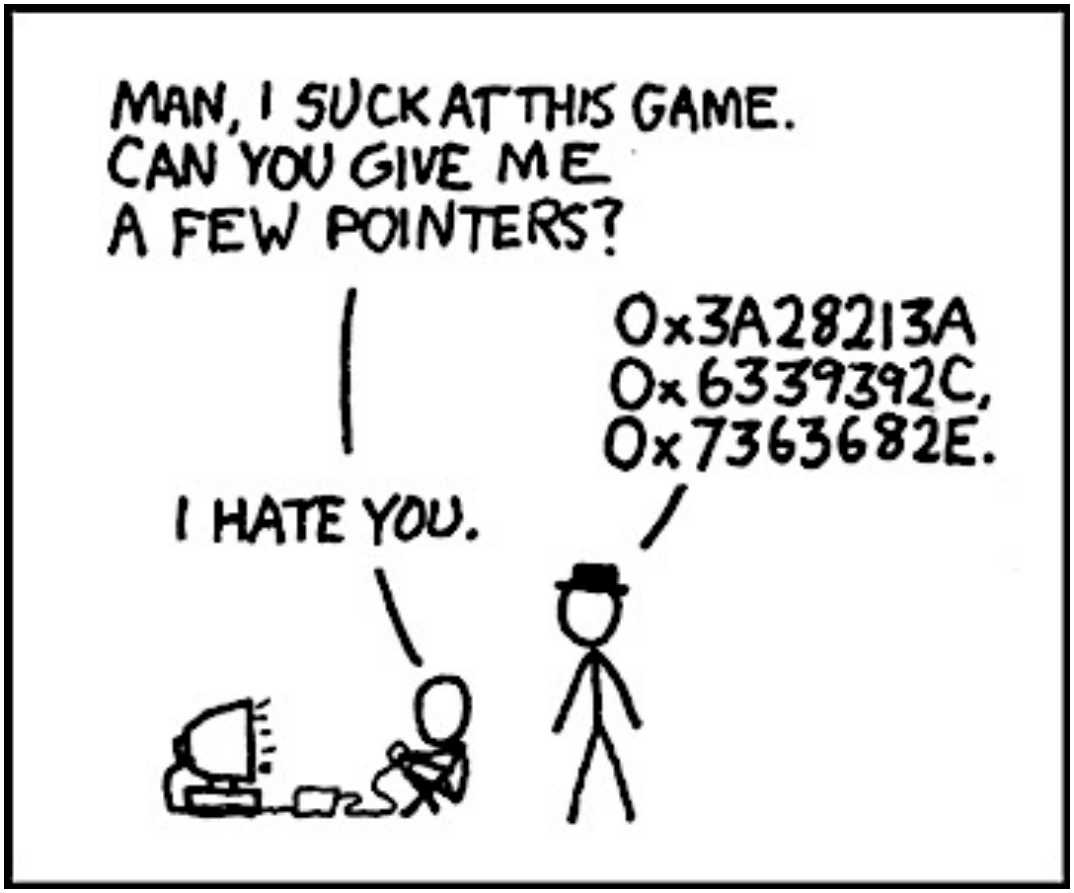
COEN 79
Object-Oriented Programming and Advanced Data Structures
Department of Computer Engineering, Santa Clara University
Behnam Dezfouli, PhD

Pointers and Arrays

Slide Set: 4

Learning Objectives

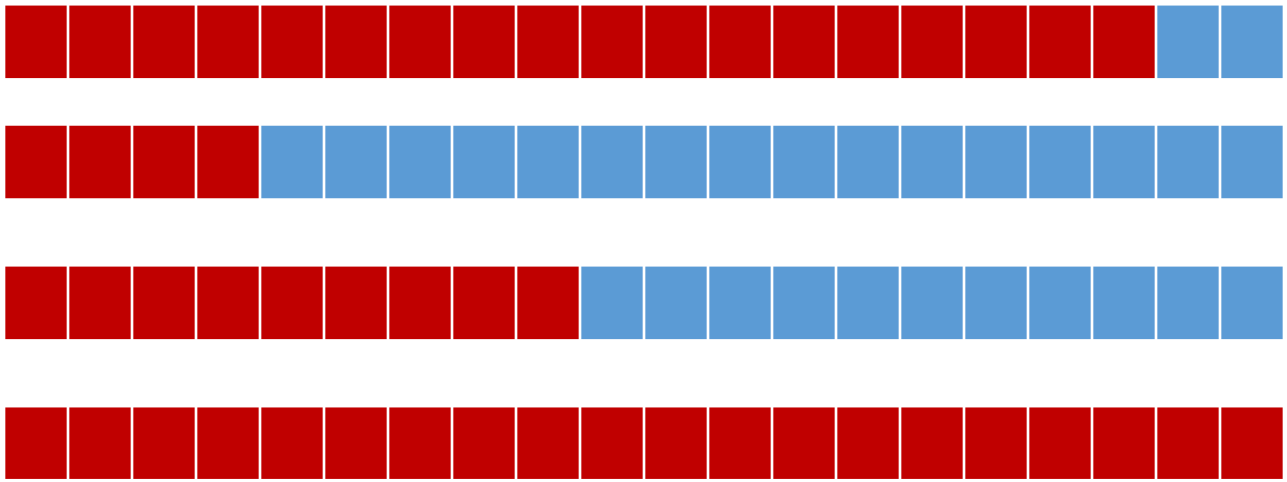
- Pointers
- Use pointer variables to allocate **dynamic memory**
- Difference between **heap** and **stack** memory
- **Releasing** dynamically allocated memory
- Pointers and arrays **as parameters** to functions
- Implementation of **dynamic data structures** with dynamic arrays
  - Implementing the bag class with dynamic array
- The string class with dynamic array
- A data structure for polynomials
- Related projects: the string class, and the polynomial class


MAN, I SUCK AT THIS GAME.
CAN YOU GIVE ME
A FEW POINTERS?
0x3A28213A
0x6339392C,
0x7363682E.
I HATE YOU.

Introduction: Why Dynamic Data Structures?

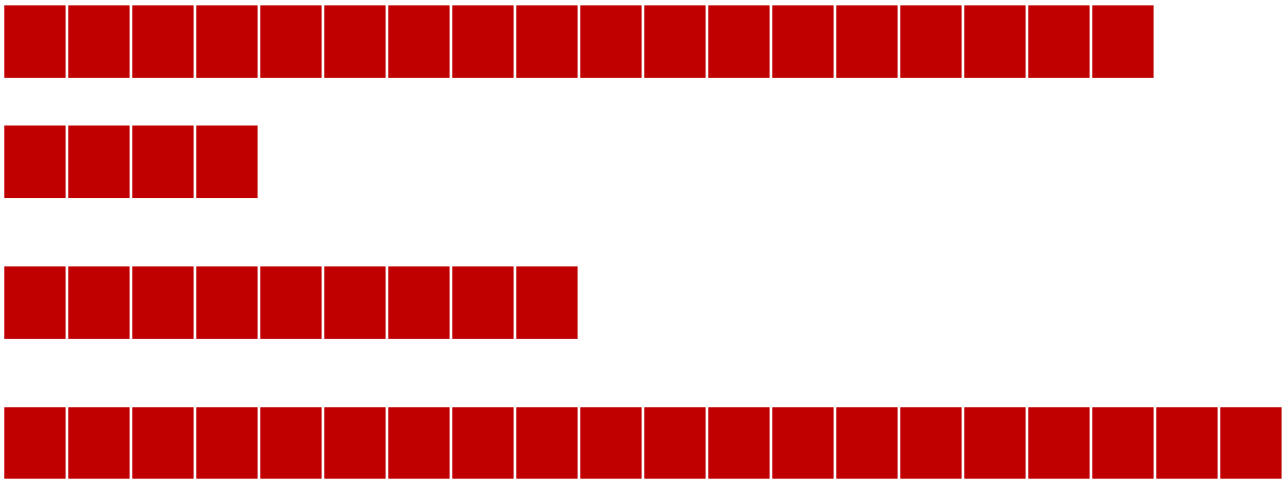
Introduction

- The container classes' capacity is declared as a **constant** in the class definition (`bag::CAPACITY`)
- **If we need bigger bags, then we can increase the constant and recompile the code**
- What if a program needs one large bag and many small bags?
- All the bags will be of the same size!



Solution:

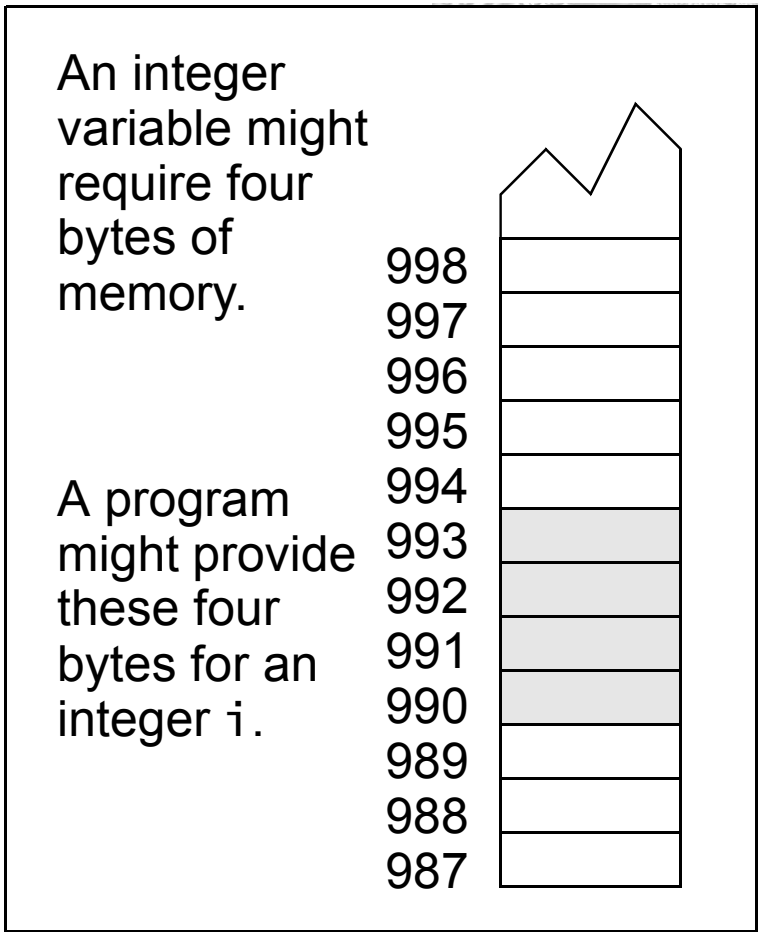
- Provide control over the size of each bag, independent of the other bags
- This control can come from **dynamic arrays**:
  - Arrays whose size is determined while a program is **actually running (not at compile time)**



Pointers and Dynamic Memory

Pointers

- **Pointer is the memory address of a variable**
- The numbers labeling each byte are called the **memory addresses**
- **When a variable occupies several adjacent bytes, the memory address of the first byte is called the memory address of the variable**
- The address of a variable is called a **pointer**



Pointer Variables

- **Pointer variable** must be declared by placing an **asterisk** before the pointer variable's name:

```
Type_Name *var_name1;
```

The type of data that the pointer variable can point to

An asterisk

The name of the newly-declared pointer variable

❑ **Example:**

```
double *my_first_ptr;
```

- `my_first_ptr` can hold the memory address of a double variable

Pointer Variables (Cont'd)

- Assignment statement:

```
int *example_ptr;
int i;

example_ptr = &i;
```

- **This statement puts the address of `i` into the pointer variable `example_ptr`**
- **So `example_ptr` now "points to" `i`**

- **`&` operator:** Is called the **address operator**, and provides the address of a variable

❑ Example: `&i` is "the address of the integer variable `i`"

- Two ways to refer to `i`:
  - You can call it `i`, or
  - You can call it "the variable pointed to by `example_ptr`"

Pointer Variables (Cont'd)

- In C++ "the variable pointed to by `example_ptr`" is written `*example_ptr`
- This is the same asterisk notation that we used to declare `*example_ptr`, **but now it has yet another meaning**
- When the asterisk is used in this way, it is called the **dereferencing operator**, and the pointer variable is said to be **dereferenced**

Pointer Variables (Cont'd)

- Example:

```
int *example_ptr;
int i;

i = 42;
example_ptr = &i;

cout << &i << endl;
cout << example_ptr << endl;

cout << i << endl;
cout << *example_ptr << endl;
```

This dereferences `example_ptr`

```
Output:
0x7fff5fbff574
0x7fff5fbff574
42
42
```

Pointer Variables (Cont'd)

- The implementation of a **reference parameter** is accomplished by using the **address** of the actual argument, rather than making a completely separate copy (as a value parameter does)

```
student object1;

student *ptr;

ptr = &object1;
```

How the memory looks like
after these statements?

Using the Assignment Operator with Pointers

- You can copy the value of one pointer variable to another with the usual assignment operator

❑ Example

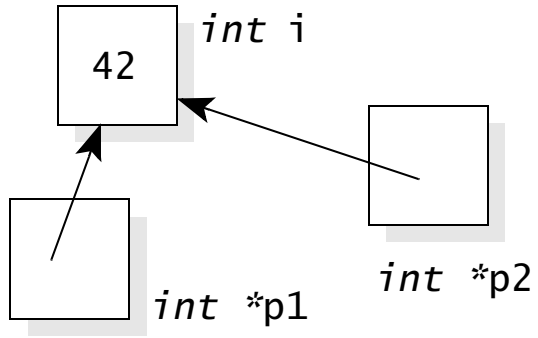
```
int i = 42;
int *p1;
int *p2;

p1 = &i;
p2 = p1;

cout << *p1 << endl;
cout << *p2 << endl;
```

p1 now points to i

p2 also points to i

After the statements:

```
p1 = &i;
p2 = p1;
```



Using the Assignment Operator with Pointers (Cont'd)

- There is a critical distinction between a pointer variable (such as p1) and the thing it points to (such as ∗p1)

```
p2 = p1          versus          *p2 = *p1
```

- What is the difference?

Dynamic Variables and the new Operator

- Real power of pointers arises when pointers are used with special kinds of variables called **dynamically allocated variables**, or more simply, **dynamic variables**
- Dynamic variables are like ordinary variables, with two important differences:
  - **They are not declared**
  - **They are created during the execution of a program**
- To create a dynamic variable while a program is running, C++ programs use an operator called **new** (declared in the global namespace)

Dynamic Variables and the new Operator (Cont'd)

❑Example:

- The creation of new dynamic variables is called **memory allocation** and the memory is **dynamic memory**
- We may say that "**`d_ptr` points to a newly allocated double variable from dynamic memory**"

```
double *d_ptr;

d_ptr = new double;
```

How the memory looks like after these statements?

`new` operator creates a new dynamic variable of type double and returns a pointer to this new dynamic variable

Using new to Allocate Dynamic Arrays

- **new** can allocate an entire array at once, the number of array components is listed in square brackets, immediately after the component data type

❑Example

```
double *d_ptr;

d_ptr = new double[10];
```

The new operator allocates an array of 10 double components and points d_ptr to the first component

- When **new** allocates an entire array, it actually **returns a pointer to the first component of the array**

Using new to Allocate Dynamic Arrays (Cont'd)

- Example:

```
int *p1;
```

```
p1 = new int[4];
```

```
p1[2] = 3;
```

- How the memory looks like after these statements?

Different versions of new Operator

```
class box {
public:
    box() { weight = 0;};
    box(int weight)
        {this->weight = weight;};
    double getWeight ()
        {return weight;};

private:
    double weight;
};

int main() {
   box *box_ptr1;
   box_ptr1 = new box;

   box *box_ptr2;
   box_ptr2 = new box (30);
   return 0;
}
```

- **If the dynamic variable is an object of a class, then the default constructor will be called to initialize the new class instance**
- A different constructor will be called if you place the constructor's arguments after the type name in the `new` statement

Different versions of new Operator (Cont'd)

- The array version of `new` is particularly useful because the number of array components can be calculated while the program is running

- If the data type of the **array** component is a **class**, then the **default constructor** is used to initialize all components of the dynamic array

```
fruit *fruit_ptr;
t_ptr = new fruit [100];
```

- The number of components can depend on factors such as user input
- This is **dynamic behavior**—behavior that is determined when a program is **running**

❑ Example of Dynamic Behavior

- Write a program that reads a list of numbers and computes the average of the numbers
- **The number of items is unknown while you develop the program**

❑**Solution**?

The Heap and the `bad_alloc` Exception

- When **`new`** allocates a dynamic variable or dynamic array, the memory comes from a location called the program's **heap** (also called the **free store**)
- Even the largest heap can be exhausted by allocating too many dynamic variables, **when the heap runs out of room, the `new` operator fails**
- The **`new`** operator usually indicates failure by throwing an exception called the **`bad_alloc`** exception
- Normally, an exception causes an error message to be printed and the program to halt
- Alternatively, a programmer can "catch" an exception and try to fix the problem

Exceptions

- **Exceptions** provide a way to react to exceptional circumstances (like runtime errors) in programs
- When an exception is thrown, control is transferred to its **handler**

Exceptions (Cont'd)

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

int main () {
    int input;
    cout << "what is the input? "<< '\n';
    cin >> input;

    try
    {
        if (input < 20)
            cout << "nice number!" << '\n';
        else
          throw 20;
    }
    catch (int e)
    {
          cout << "An exception occurred. Exception#: " << e << '\n';
    }
    return 0;
}
```

```
Terminal:
what is the input?
20
An exception occurred.
Exception#: 20
```

Exceptions (Cont'd)

```
// bad_alloc example
#include <iostream>      // std::cout
#include <new>           // std::bad_alloc

int main () {

    try
    {
        int* myarray= new int[1000000];
    }

   catch (std::bad_alloc& ba)
    {
        std::cerr << "bad_alloc caught: " << ba.what() << '\n';
    }

    return 0;
}
```

If memory allocation is unsuccessful, then the output will be:
`bad_alloc caught: bad allocation`

The `delete` Operator

- The size of the heap varies from one computer to another, it could be just a few thousand bytes or more than a billion
- **Even with small programs, it is an efficient practice to release any heap memory that is no longer needed**
- The **`delete`** operator is used to return the memory of a dynamic variable back to the heap where it can be reused for more dynamic variables

❑Example

```
int *example_ptr;
example_ptr = new int;

…

delete example_ptr;
```

Clarifying Heap and Stack Memory

- **The Stack Memory**
- A special region of memory that stores temporary variables created by each function (including the `main()` function)
  - When a function declares a new variable, it is "pushed" onto the stack
  - **When a function exits, all of the variables pushed onto the stack by that function, are freed**
  - Once a stack variable is freed, that region of memory becomes available for other stack variables
- **Stack variables only exist while the function that created them is running**
- **Advantage: There is no need to manage the memory yourself, variables are allocated and freed automatically**

Clarifying Heap and Stack Memory (Cont'd)

Stack overflow is the result of:

- Allocating too many variables on the stack
- Making too many nested function calls
  - ❑ Example: Where function A calls function B calls function C calls function D …

- Stack overflow generally causes a program to crash

❑ Example:

```
int main()
{
    int array[100000000];
    return 0;
}
```

This program will likely cause a stack overflow

Clarifying Heap and Stack Memory (Cont'd)

- **The Heap Memory**
- A region of memory that is not managed automatically for you, and is not as tightly managed by the CPU
- Once you have allocated memory on the heap, you are responsible for releasing that memory
  - If you fail to do this, your program will have what is known as a **memory leak**
- When you use the `new` operator to allocate memory, this memory is allocated in the program's heap segment
- Scope:
  - Variables created on the heap are accessible by any function, anywhere in your program (unlike stack)
  - **Heap variables are essentially global in scope**

The `delete` Operator (Cont'd)

- `delete` operator can also free a dynamic array of components
- To free an entire array, the array brackets [ ] are placed after the word `delete`

❑ Example

```
int *example_ptr;
example_ptr = new int[50];
…

delete [ ] example_ptr;
```

Define Pointer Types

- You can define a name for a pointer type
- Enables you to declare pointer variables without using asterisk

❑ Example:

```
typedef int* int_pointer;
```

Defines a data type called `int_pointer`, which is the type for pointer variables that point to `int` variables

- A type definition such as this usually appears in a header file or with the collection of function prototypes that precede a main program

- The following declarations are equal:

`int_pointer i_ptr;` is equivalent to **`int *i_ptr;`**

Pointers and Arrays as Parameters

Pointers and Arrays as Parameters

Value Parameters that are Pointers

- When a **value parameter is a pointer**, the function **may change the value in the location that the pointer points to**
- The actual argument in the calling program will still point to the same location, but that location will have a new value

❑ Example:

```
void make_it_3(int* i_ptr)
{
// Precondition: i_ptr is pointing to an integer variable.
// Postcondition: The integer that i_ptr is pointing at has been
// changed to 3.
    *i_ptr = 3;
}
```

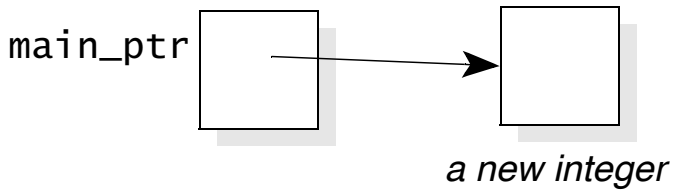
- Parameter `i_ptr` has type `int*`, that is, a pointer to an integer, **and it is a value parameter**
- The body of the function does not actually change `i_ptr`; it changes only the integer that `i_ptr` points to

Pointers and Arrays as Parameters

Value Parameters that are Pointers (Cont'd)

- Example: Assume a program declares a pointer to an integer, allocating memory for the pointer to point to, and calls `make_it_3`:

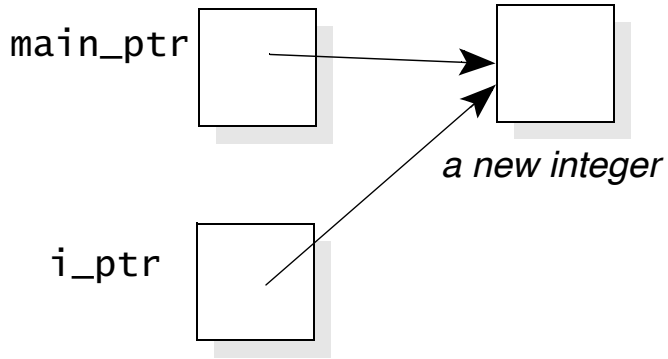
```
int *main_ptr;
main_ptr = new int;
```



Now `main_ptr` is pointing to a newly allocated integer

```
make_it_3(main_ptr);
```

- **The argument, `main_ptr`, provides the initial value for the parameter `i_ptr`**

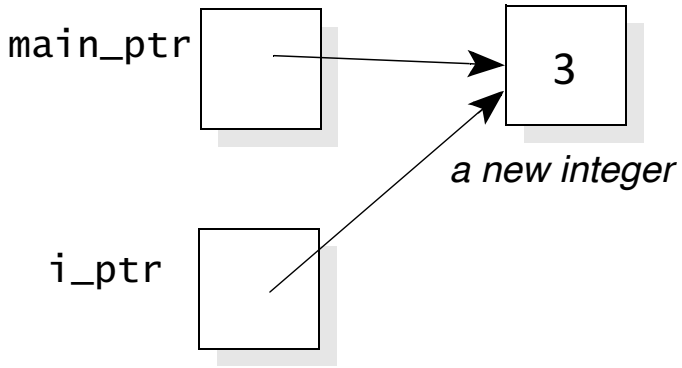


Pointers and Arrays as Parameters

Value Parameters that are Pointers (Cont'd)

- In the body of `make_it_3` function, the assignment statement `*i_ptr = 3` places 3 in the location that `i_ptr` points:

```
void make_it_3(int* i_ptr)
{
        *i_ptr = 3;
}
```



- When the function returns, the formal parameter `i_ptr` is freed because it is a local variable allocated on stack memory
- However, the pointer variable `main_ptr` is still around, and it is still pointing to the same location, but the location has a new value of 3

Pointers and Arrays as Parameters

Array Parameters

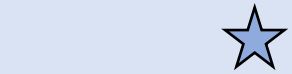
- **When a parameter is an array, it is automatically treated as a pointer that points to the first element of the array**

❑Example

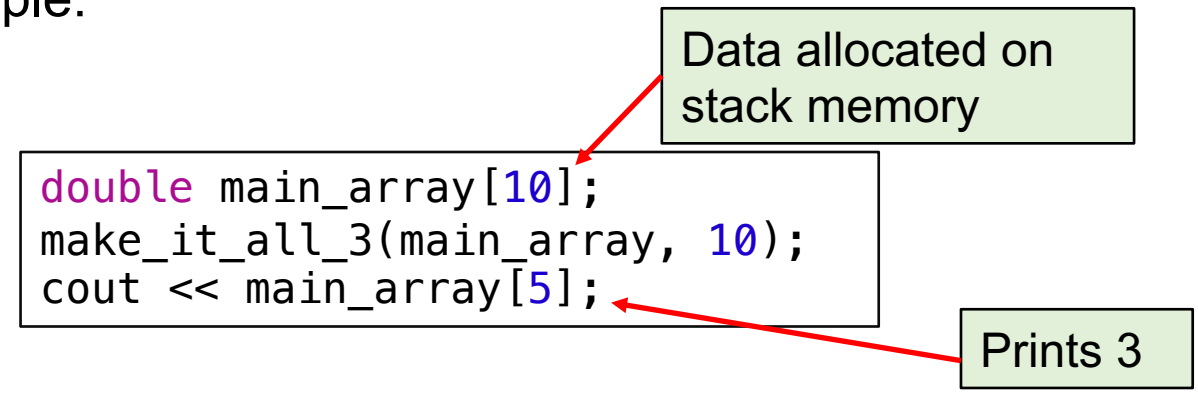
```
void make_it_all_3(double data[ ], size_t n)
// Precondition: data is an array with at least n components.
// Postcondition: The first n elements of the data have been set to 3.
{
    size_t i;
    for (i = 0; i < n; ++i)
        data[i] = 3;
}
```

- If the body of the function changes the components of the array, the changes **do** affect the actual argument

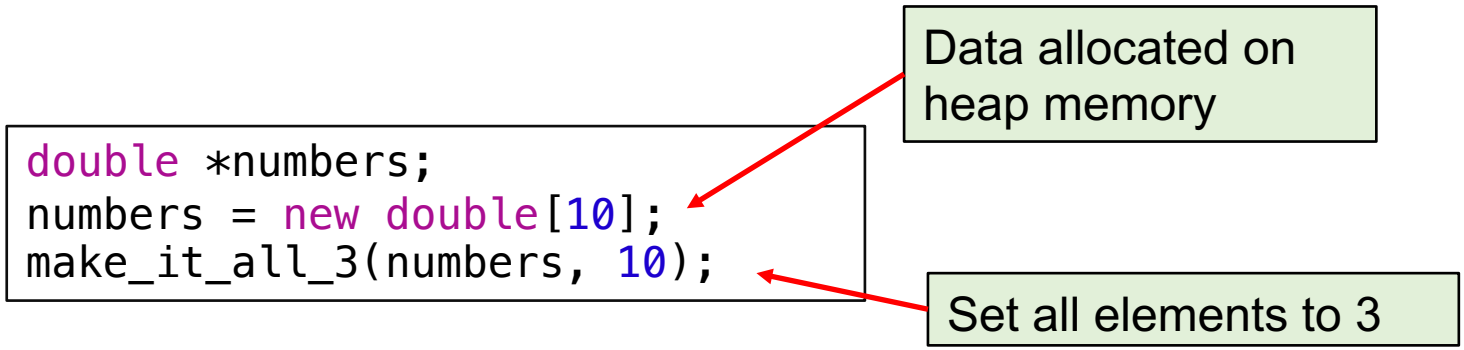
Array Parameters (Cont'd)



❑ Example:



- The actual argument of `make_it_all_3` may be a dynamic array:



Pointers and Arrays as Parameters

`const` Parameters that are Pointers or Arrays

- A parameter that is a pointer may also include the `const` keyword

❑ Example

- **`const` keyword indicates that `i_ptr` is a pointer to a constant integer**
- **Implementation of `is_3` may examine `*i_ptr`, but may not change the value of `*i_ptr`**

```
bool is_3(const int* i_ptr);
```

```
double average(const double data[ ], size_t n);
```

`const` keyword indicates that the function cannot change the array entries

- Note: The functions may examine the item that is pointed to (or the array), but changing the item (or array) is forbidden

Pointers and Arrays as Parameters

`const` Parameters that are Pointers or Arrays (Cont'd)

- Example

```
bool is_3(const int* i_ptr)
{
   // Precondition: i_ptr is pointing to an integer variable.
   // Postcondition: The return value is true if *i_ptr is 3.
   return (*i_ptr == 3);

}
```

- It just examines the `*i_ptr`
- It does not change the `*i_ptr`

Pointers and Arrays as Parameters

const Parameters that are Pointers or Arrays (Cont'd)

- Example:

```
double average(const double data[ ], size_t n)
// Library facilities used: cassert, cstdlib
{
   size_t i;     // An array index
   double sum;   // The sum of data[0] through data[n - 1]

   assert(n > 0);

   // Add up the n numbers and return the average.
   sum = 0;

   for (i = 0; i < n; ++i)
     sum += data[i];


    return (sum/n);
}
```

It examines all the array entries, but it does not change them

Reference Parameters that are Pointers

A reference parameter that is a pointer is used when a function:

- **Changes a pointer parameter so that the pointer points to a new location**

And

- **The programmer needs the change to affect the actual argument**

Pointers and Arrays as Parameters

Reference Parameters that are Pointers

- Example

p is a pointer to a `double` (that is, `double*`) and it is a reference parameter (indicated by the symbol &)

```
void allocate_doubles(double*& p, size_t& n)
// Postcondition: The user has been prompted for a size n, and this
// size has been read.
// The pointer p has been set to point to a new dynamic array
// containing n doubles.
// NOTE: If there is insufficient dynamic memory, then bad_alloc is
// thrown.
{
     cout << "How many doubles should I allocate?" << endl;
     cout << "Please type a positive integer answer: ";
     cin >> n;
     p = new double[n];
}
```

- Allocates the array of n doubles
- p points to the dynamically allocated memory

Pointers and Arrays as Parameters

Reference Parameters that are Pointers (Cont'd)

```
void allocate_doubles(double*& p, size_t& n);
```

- In a program, we can use `allocate_doubles` to allocate an array of double values, with the size of the array determined by interacting with the user

❑ Example

```
double *numbers;
size_t array_size;
allocate_doubles(numbers, array_size);
```

Look at Appendix 1: Demonstration program for the dynamic array

Pointers and Arrays as Parameters

Reference Parameters that are Pointers (Cont'd)

```
#include <iostream>

void function_a(int *& a)
{
    *a += 5;
    int* c = new int(7);
    a = c;
}

void function_b(int * a)
{
    *a += 5;
    int* c = new int(7);
    a = c;
}
```

```
int main()
{
    int* myInt = new int(5);
    int* myInt2 = new int(5);

    function_a(myInt);
    std::cout << myInt << std::endl;
    std::cout << *myInt << std::endl;

    function_b(myInt2);
    std::cout << myInt2 << std::endl;
    std::cout << *myInt2 << std::endl;

    return 0;
}
```

```
Output:
0x100202210
7
0x100200570
10
```

Discuss about the output…

Pointers and Arrays as Parameters

Overview of Parameter Types

- Value Parameter

```
void function(double p);
```

- Reference Parameter

```
void function(double& p);
```

- const Reference Parameter

```
void function(const double& p);
```

- Value Parameter that is Pointer

```
void function(double* p);
```

- const Value Parameter that is Pointer

```
void function(const double* p);
```

- Reference Parameter that is Pointer

```
void function(double*& p);
```

The Bag Class with Dynamic Arrays

The Bag Class with Dynamic Arrays

- Pointers enable us to define data structures whose size is determined when a program is actually **running** rather than at **compilation time**
- Such data structures are called **dynamic data structures**
- This is in contrast to **static data structures**, which have their size determined when a program is **compiled**
- A class may be a dynamic data structure, i.e., it may use dynamic memory

The Bag Class with Dynamic Arrays

Pointer Member Variables

- The original bag class has a member variable that is a **static array** containing the bag's items
- Our dynamic bag has a member variable that is a pointer to a **dynamic array**

The Static Bag:

```
class bag
{
    ...
private:
    value_type  data[CAPACITY];
    size_type  used;
};
```

The Dynamic Bag:

```
class bag
{
    ...
private:
    value_type  *data;
    size_type  used;
};
```

The Bag Class with Dynamic Arrays

Pointer Member Variables (Cont'd)

- The constructor for the dynamic bag will allocate a dynamic array
- As a program runs, a new, larger dynamic array can be allocated when we need more capacity

```
class bag
{
  public:
    ...
  private:
    value_type *data;
    size_type used;
    size_type capacity;
};
```

Points to a **partially filled dynamic array** that stores the actual items of the bag

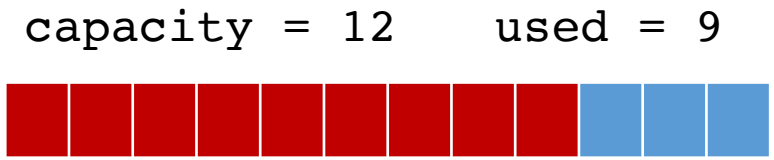
Stores the number of items in the bag

Stores the total size of the dynamic array

The Bag Class with Dynamic Arrays

Pointer Member Variables (Cont'd)

- The capacity and used portions of the bag may change over time



Member Functions Allocate Dynamic Memory As Needed

- When a class uses dynamic memory, **the class's member functions allocate dynamic memory as needed**

❑ Example

- The constructor of the dynamic bag allocates the dynamic array that the member variable data points to
  - **Question**: How big should this array be?
  - Our plan is to have the constructor allocate a dynamic array whose initial size is determined by a parameter to the constructor
  - Whenever items are inserted into a bag (through the `insert` member function or the `+= operator`) the bag's capacity may be increased

Member Functions Allocate Dynamic Memory As Needed (Cont'd)

- **Why a programmer needs to be concerned about the initial capacity of a bag?**
- Can't we just start with a small initial capacity and insert items one after another? The `insert` function will take care of increasing the capacity as needed
  - Yes, this approach works correctly
  - However, if there are many items, then **many of the activations of `insert` would need to increase the capacity, and this could be inefficient**
  - **Each time the capacity is increased, new memory is allocated, the items are copied into the new memory, and the old memory is released**
  - **To avoid this repeated allocation of memory**, a programmer can request a large initial capacity

Member Functions Allocate Dynamic Memory As Needed (Cont'd)

- The new bag's constructor:

```
bag(size_type initial_capacity = DEFAULT_CAPACITY);
// Postcondition: The bag is empty with a capacity given by the
// parameter.
// The insert function will work efficiently (without allocating
// new memory) until this capacity is reached.
```

❑Example

- When the bag is declared, the programmer can specify a capacity of 1000:

```
bag kilosack(1000);
```

1000 items can be efficiently added to `kilosack`

- After the initial capacity is reached, the `insert` function continues to work **correctly**, but it might be **slowed down by memory allocations**

Member Functions Allocate Dynamic Memory As Needed (Cont'd)

```
bag(size_type initial_capacity = DEFAULT_CAPACITY);
```

- Notice that the parameter of the constructor has a **default argument**, `DEFAULT_CAPACITY`, which will be a constant in our class definition
- **The single constructor actually serves two purposes**:
  - It **can be used with an argument** to construct a bag with a specific capacity, or
  - It **can be used as a default constructor** (with no argument list)

❑Example:

```
bag ordinary;
bag super(9000);
```

The initial capacity is `DEFAULT_CAPACITY`

The initial capacity is 9000

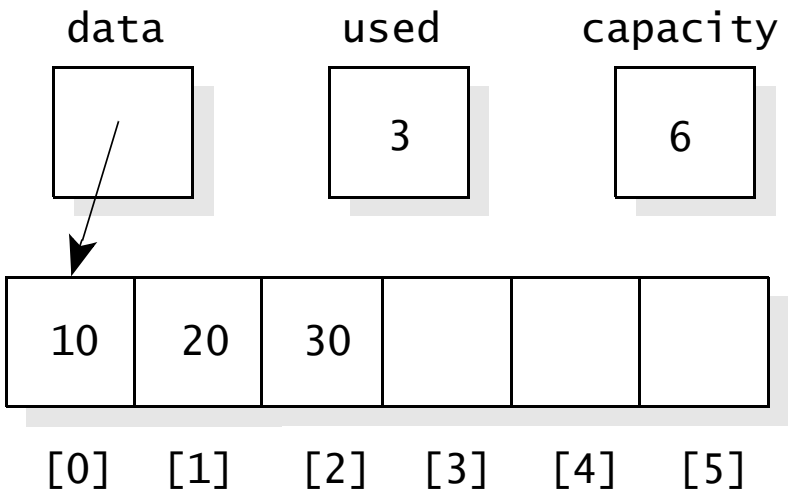
Member Functions Allocate Dynamic Memory As Needed (Cont'd)

- Example

```
bag sixpack(6);

sixpack.insert(10);
sixpack.insert(20);
sixpack.insert(30);
```

The constructor creates a bag with an initial capacity of 6

- After these declarations, the bag's private member variables look like this:



Member Functions Allocate Dynamic Memory As Needed (Cont'd)

- While the bag is in use, a programmer can make an explicit adjustment to the bag's capacity via a member function called **`reserve`**:

```
void reserve(size_type new_capacity);
// Postcondition: The bag's current capacity is changed to the
// new_capacity (but not less than the number of items already in
// the bag).
// The insert function will work efficiently (without allocating
// new memory) until the new capacity is reached.
```

- By using the `reserve` member function, the bag's efficiency is improved
- The constructor, `reserve`, `insert`, `operator+=` and `operator+` member functions can allocate new dynamic memory

The Bag Class with Dynamic Arrays

Documentation for the Dynamic Bag Header File

```
// ……….
//
// NONMEMBER FUNCTIONS for the bag class:
// bag operator +(const bag& b1, const bag& b2)
// Postcondition: The bag returned is the union of b1 and b2.
//
//
// VALUE SEMANTICS for the bag class:
// Assignments and the copy constructor may be used with bag objects.
//
//
// DYNAMIC MEMORY USAGE by the bag:
// If there is insufficient dynamic memory, then the following
// functions throw bad_alloc: The constructors, reserve, insert,
// operator += , operator +, and the assignment operator.
```

This documentation allows experienced programmers to deal with potential failure

Provide Documentation About Possible Dynamic Memory Failure

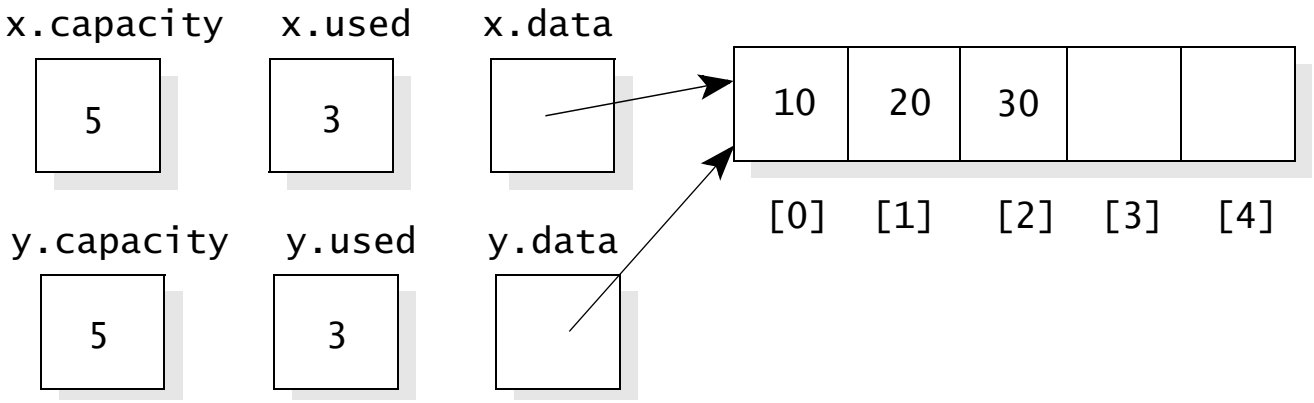
- There are two extra factors that play an important role whenever a class uses dynamic memory:
  - The first factor is the **value semantics** (i.e., the assignment operator and the copy constructor)
  - The second factor is a requirement for a special member function called a **destructor**

Value Semantics

- With all our other classes**, it was sufficient to use the automatic assignment operator** and the **automatic copy constructor**
  - In the past, when we wrote y = x, we were content to let the automatic assignment operator copy all the member variables from the object x to the object y
- **The automatic assignment operator fails for the dynamic bag (or for any other class that uses dynamic memory)**

Value Semantics (Cont'd)

❑ Example: Suppose we set up a bag called `x` with an initial capacity of 5, containing the integers 10, 20, and 30—then we assign `x` to bag `y` through `y = x`

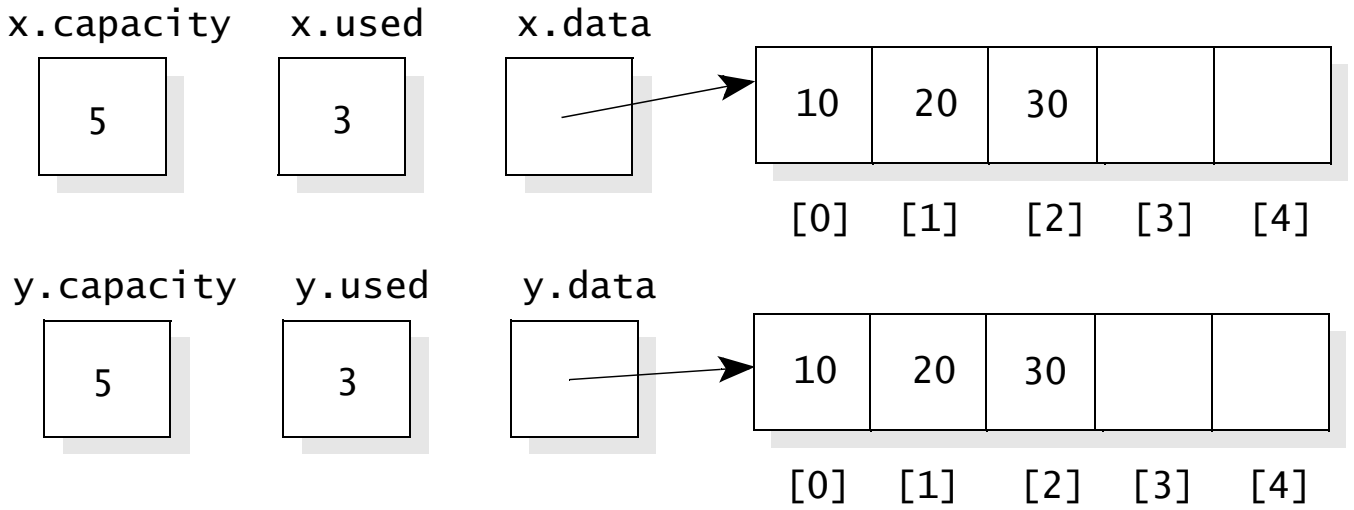


- What is the problem?

The Bag Class with Dynamic Arrays

Value Semantics (Cont'd)

- When we assign `y = x`, we want `y` to have its own dynamic array, completely separate from `x`'s dynamic array



- **We must provide our own assignment operator** rather than relying on the automatic assignment operator
- **We can do this by overloading the assignment operator** for the bag class

The Bag Class with Dynamic Arrays

Value Semantics (Cont'd)

```
void bag::operator =(const bag& source);
// Postcondition: The bag that activated this function
has the // same items and capacity as source.
```

- When you overload the assignment operator, **C++ requires it to be a member function**
- In an assignment statement `y = x`, in fact we have `y.operator=(x)`, thereby **bag `y` is activating the function**, and the bag `x` is the argument for the parameter named `source`

Value Semantics (Cont'd)

- The second part of the value semantics is the **copy constructor**, which is activated **when a new object is initialized as a copy of an existing object**
- Unless you indicate otherwise, y is initialized using the automatic copy constructor, which merely copies the member variables from x to y
- If you want to avoid the simple copying of member variables, then **you must provide a copy constructor with the prototype:**

```
bag::bag(const bag& source);
// Postcondition: The bag that is being constructed has been
// initialized with the same items and capacity as source.
```

- Note: The parameter of the copy constructor is usually a const reference parameter (C++ also permits an ordinary reference parameter, **but does not allow a value parameter)**

The Destructor

- The **destructor** of a class is **a member function that is automatically activated when an object becomes inaccessible**
- **The primary purpose of the destructor is to return an object's dynamic memory to the heap when the object is no longer in use**
- The destructor has three unique features:
  1. The name of the destructor is always the tilde character (~) followed by the class name
  2. The destructor has no parameters and no return value
     - Example: `~bag( );`
  3. Programmers who **use** a class should not need to know about the destructor
     - Programs rarely activate the destructor explicitly
     - The activation is usually **automatic** whenever an object becomes inaccessible

The Destructor (Cont'd)

- Several common situations cause automatic destructor activation:
  1. When **a local variable is an object with a destructor**, the destructor is **automatically activated when the function returns**

     ❑Example

```
void example1( )
{
     bag sample1;
     ...
}
```

     - When the function `example1` returns, the destructor `sample1.~bag( )` is automatically activated

The Destructor (Cont'd)

2. Suppose **a function has a value parameter that is an object**

```
void example2(bag sample2)
// Does some calculation using a bag
```

- When the function `example2` returns, the destructor `sample2.~bag( )` is automatically activated

- Note: If `sample2` was a reference parameter, then the destructor would not be activated because a reference parameter is actually an object in the calling program, **and that object is still accessible**

The Destructor (Cont'd)

3. Suppose that **a dynamic variable is an object**

```
bag *b_ptr;
b_ptr = new bag;
...
delete b_ptr;
```

- When `delete b_ptr` is executed, the destructor for `*b_ptr` is automatically activated
- **The destructor ensures that the dynamic array used by `*b_ptr` is released**

The Bag Class with Dynamic Arrays

Header File for the Bag Class with a Dynamic Array

```
// FILE: bag2.h (part of the namespace scu_coen79_4)
// CLASS PROVIDED: bag
#ifndef SCU_COEN79_BAG2_H
#define SCU_COEN79_BAG2_H
#include <cstdlib>   // Provides size_t

namespace scu_coen79_4
{
    class bag
    {
    public:
        // TYPEDEFS and MEMBER CONSTANTS
        typedef int value_type;
        typedef std::size_t size_type;
        static const size_type DEFAULT_CAPACITY = 30;

        // CONSTRUCTORS and DESTRUCTOR
        bag(size_type  initial_capacity = DEFAULT_CAPACITY);
        bag(const bag& source);
        ~bag( );
```

Initializing an empty bag with an initial capacity "DEFAULT_CAPACITY"

Destructor

copy constructor

The Bag Class with Dynamic Arrays

Header File for the Bag Class with a Dynamic Array (Cont'd)

```
// MODIFICATION MEMBER FUNCTIONS
```

Increases the bag's current capacity to the `new_capacity`

```
void reserve(size_type new_capacity);
bool erase_one(const value_type& target);
size_type erase(const value_type& target);
```

Inserts a new copy of `entry` to the bag

```
void insert(const value_type& entry);
void operator +=(const bag& addend);
```

Enables a bag to have same items and capacity as `source`

```
void operator =(const bag& source);

// CONSTANT MEMBER FUNCTIONS
size_type size( ) const { return used; }
size_type count(const value_type& target) const;
```

Header File for the Bag Class with a Dynamic Array (Cont'd)

```
private:
        // Pointer to partially filled dynamic array
        value_type *data;

        // How much of array is being used
        size_type used;

        // Current capacity of the bag
        size_type capacity;
};

    // NONMEMBER FUNCTIONS for the bag class
    bag operator +(const bag& b1, const bag& b2);
}

#endif
```

The Revised Bag Class — Implementation

- Three functions are particularly important:
  1. **Constructors**
  2. **Destructor**
  3. **Assignment operator**

- **These three member functions are always needed when a class uses dynamic memory**

The Revised Bag Class — Implementation (Cont'd)

- **The constructors:** Each of the constructors is responsible for setting up the three private member variables in a way that satisfies the invariant of the dynamic bag class

❖**Constructor**

Tells how many items to allocate for the dynamic array

```
bag::bag(size_type initial_capacity)
    {
        data = new value_type[initial_capacity];
        capacity = initial_capacity;
        used = 0;
    }
```

The Revised Bag Class — Implementation (Cont'd)

❖**Copy constructor**

```
bag::bag(const bag& source)
    // Library facilities used: algorithm
    {
        data = new value_type[source.capacity];
        capacity = source.capacity;
        used = source.used;
        copy(source.data, source.data + used, data);
    }
```

start **end** **destination**

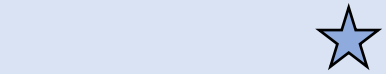
- The capacity of the dynamic array is the same as the capacity of the bag that is being copied
- After the dynamic array has been allocated, the items may be copied into the newly allocated array

The Revised Bag Class — Implementation (Cont'd)

3. The destructor: The primary responsibility of the destructor is **releasing dynamic memory**

```
bag::~bag( )
    {
        delete [ ] data;
    }
```

The Revised Bag Class — Implementation (Cont'd)



- **The assignment operator**
- Assignment operator vs. Copy constructor:
  - The assignment operator is not constructing a new bag, meaning that there is already a partially filled array allocated
    - **The size of this array might need to be changed**
    - **If we allocate a new array, then the original array must be returned to the heap**
  - In the assignment operator, it is possible that the source parameter (which is being copied) is the same object that activates the operator
    - With a `bag b`, this would occur if a programmer writes `b = b` (called a **self- assignment**)

The Revised Bag Class — Implementation (Cont'd)

The assignment operator

- The solution for the self-assignment is to provide a special check at the start of the operator
- If we find that an assignment such as `b = b` is occurring, then we will return immediately
- We can check for this condition by determining whether source is the same object as the object that activated the operator

```
// Check for possible self-assignment:
if (this == &source)
    return;
```

A **pointer** to the object that activated the function

Address of the `source` object

The Revised Bag Class — Implementation (Cont'd)

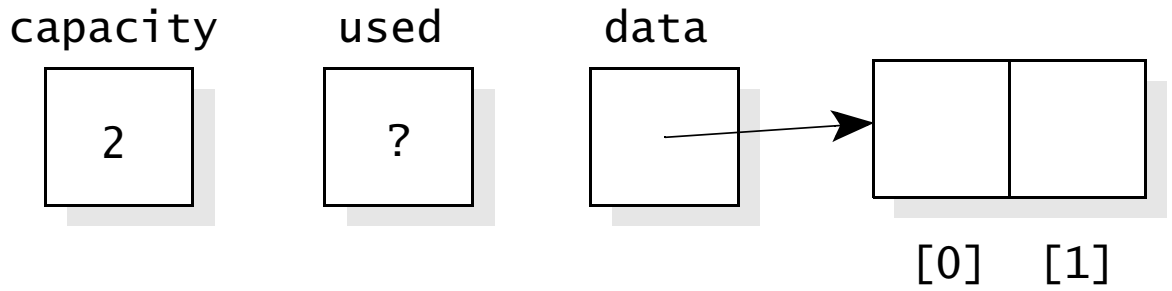
The assignment operator

```
void bag::operator =(const bag& source)
{
 value_type *new_data;

 if (this == &source)  return; // Check for self-assignment

 // If needed, allocate an array with a different size:
 if (capacity != source.capacity)
  {
    new_data = new value_type[source.capacity];

    delete [ ] data;

    data = new_data;
    capacity = source.capacity;
  }

 used = source.used;
 copy(source.data, source.data + used, data);
}
```

The Revised Bag Class — Implementation (Cont'd)

❑Example

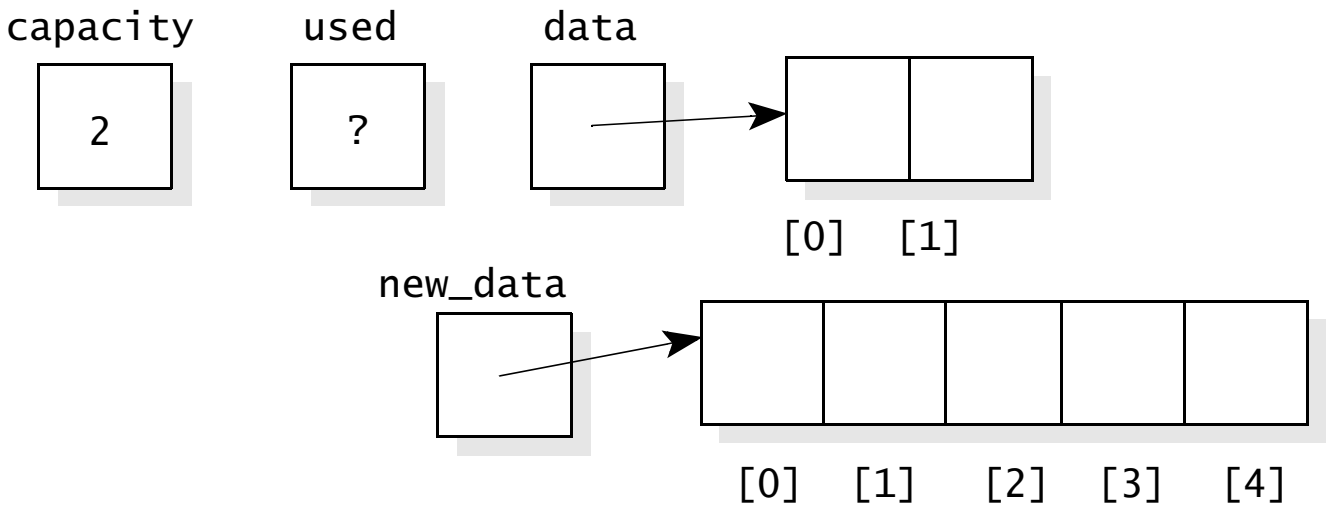
- Assume that `source` is a bag with a capacity of 5, and the bag that activated the function has a mere capacity of 2
- When the assignment begins, we have this situation:



- Since the current capacity (which is 2) is not equal to the amount needed (which is 5), the code enters the body of the if-statement

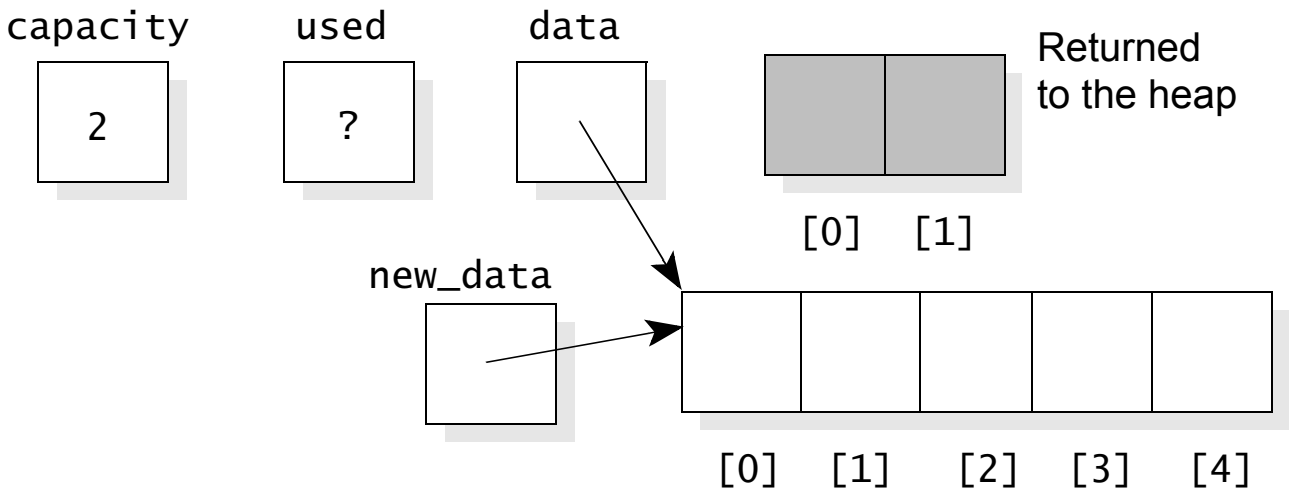
The Revised Bag Class — Implementation (Cont'd)

- In the body, we have a local variable, `new_data`, which is set to point to a newly allocated array of five items:



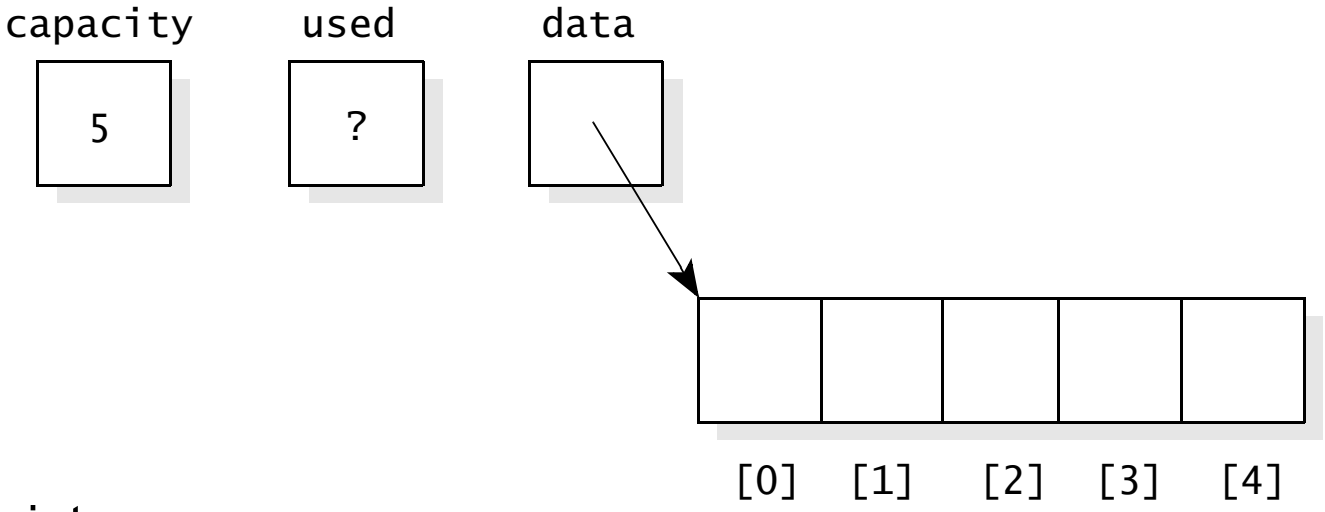
The Revised Bag Class — Implementation (Cont'd)

- Once the new array has been allocated:
  - We return the old array to the heap
  - Assign `data = new_data`, so that the data pointer points to the new array



The Revised Bag Class — Implementation (Cont'd)

- Capacity is changed to 5
- We no longer need the local variable `new_data`



- At this point:
  - Copy the items from source's array into the newly allocated array
  - Set the value of `used`

The Revised Bag Class — Implementation (Cont'd)

The assignment operator (Cont'd)

- Why not to use the following two statements for memory allocation instead of using a local variable?

```
delete [ ] data; // Release old array
data = new value_type[source.capacity]; // Allocate new array
```

The Revised Bag Class — Implementation (Cont'd)

The assignment operator (Cont'd)

- When the destructor is given an invalid object (such as our invalid bag):
  - The destructor may cause an error message that will be more confusing than the usual message from `bad_alloc`
  - Programs will be harder to debug
- The invalid bag also makes it harder for experienced programmers to deal with the `bad_alloc` exception in a way that tries to recover without halting the program
- Because of these problems: **Member functions must always ensure that all objects are valid prior to calling `new`**

The Bag Class with Dynamic Arrays

The `reserve` member function

- **`reserve` member function** is used to change the capacity of a bag

```
void bag::reserve(size_type  new_capacity)
// Postcondition: The bag's current capacity is changed to the
// new_capacity (but not less than the number of items already in
// the bag).
{
    value_type *larger_array;
    if (new_capacity == capacity)
        return; // The allocated memory is already the right size.

    if (new_capacity < used)
        new_capacity = used; // Can't allocate less than used.

    larger_array = new value_type[new_capacity];
    copy(data, data + used, larger_array);
    delete [ ] data;
    data = larger_array;
    capacity = new_capacity;
}
```

The Bag Class with Dynamic Arrays

The `insert` and `operator +=` member functions

```
void bag::insert(const value_type& entry)
    {
        if (used == capacity)
            reserve(used+1);

        data[used] = entry;
        ++used;
    }
```

```
void bag::operator +=(const bag& addend)
    // Library facilities used: algorithm
    {
        if (used + addend.used > capacity)
            reserve(used + addend.used);

        copy(addend.data, addend.data + addend.used, data + used);
        used += addend.used;
    }
```

The Bag Class with Dynamic Arrays

The `operator +` member function

```
bag operator +(const bag& b1, const bag& b2)
{


    bag answer(b1.size( ) + b2.size( ));

    answer += b1;
    answer += b2;
    return answer;
}
```

The function declares a bag of sufficient size

Prescription for a Dynamic Class

Prescription for a Dynamic Class

Four Rules

- When a class uses **dynamic memory**, follow these four rules:
  - Some of the member variables of the class are **pointers**
  - Member functions **allocate and release dynamic memory**
  - The **automatic value semantics of the class is overridden**
  - The **class has a destructor** to return all dynamic memory to the heap

Prescription for a Dynamic Class

Special Importance of the Copy Constructor

- The **copy constructor** is used when **one object is to be initialized as a copy of another**, as in the declaration:

```
bag y(x); // Initialize y as a copy of x.
```

- There are three other common situations where the copy constructor is used

1. Alternative syntax: The first situation is really just an alternative syntax for using the copy constructor to initialize **a newly declared object**:

```
bag y = x; // Initialize y as a copy of x.
```

Special Importance of the Copy Constructor (Cont'd)

2. The second situation that uses the copy constructor is **when a return value of a function is an object**

❑ Example: The bag's `operator+` **returns a bag object**

- The function computes its answer in a local variable, and then has a return statement
- When the return statement is executed, the value from the local variable is copied to a temporary location called the **return location**
- The local variable itself is then destroyed (along with any other local variables), and the function returns to the place where it was called

Special Importance of the Copy Constructor (Cont'd)

3. A third situation arises **when a value parameter is an object**

❑ Example:

```
int rotations_needed(point p);
```

- When the function is called, **the actual argument is copied to the formal parameter p**
- The copying occurs **by using the copy constructor**

Smart Pointers

Smart Pointers

- Using smart pointers, we can make pointers to work in way that **we don't need to explicitly call `delete`**
- Smart pointer **is a wrapper class over a pointer** with operator like * and -> overloaded
- The objects of smart pointer class look like pointer, but can do many things that a normal pointer can't like automatic destruction, reference counting,…

- The idea is to make a class with a pointer, destructor and overloaded operators like * and ->
- **Since destructor is automatically called when an object goes out of scope, the dynamically allocated memory is automatically deleted**

Smart Pointers

```
class smartIntPtr
{
    int *ptr;  // Actual pointer

public:
    // Constructor:
    smartIntPtr(int *p = NULL) { ptr = p; }

    // Destructor
    ~smartIntPtr() { delete ptr; }

    // Overloading dereferencing operator
    int &operator *() {  return *ptr; }
};


int main()
{
    smartIntPtr ptr( new int() );
    *ptr = 20;
    cout << *ptr << endl;

    return 0;
}
```

The STL String Class and a Project

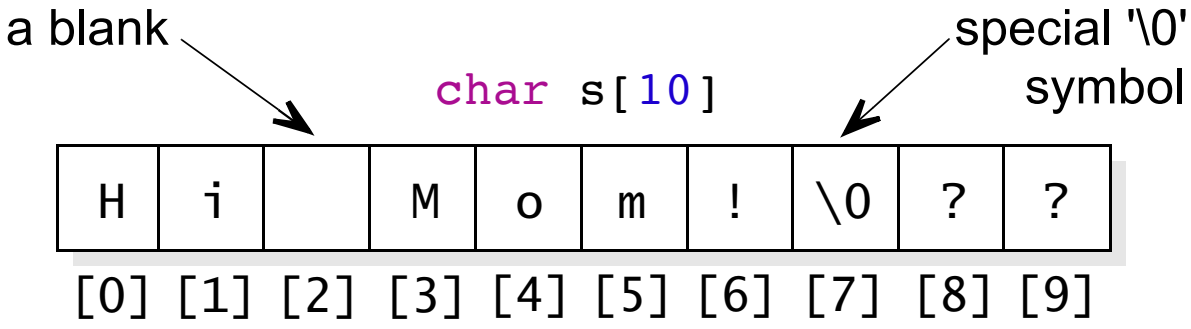
The STL String Class

Null-Terminated Strings

- In C or C++: **An array of characters** can be used **to hold a simple kind of string**
- Example: A string variable capable of storing a string **with nine or fewer characters**

```
char s[10] = { 'H', 'i', ' ', 'M', 'o', 'M', '!', '\0' };
```

- **The special symbol '\0' (null character)** is placed in the array immediately after the last character of the string



The longest possible string is one less than the size of the array

Null-Terminated Strings

- **The only distinction between a string variable and an array of characters is the fact that a string variable must use the null character to mark the end of the string**

- In a program, the null character is written `'\0'`—**the single quote marks** are used with all C++ characters such as `'a', 'b', 'c'`
  - It looks like two characters, but it is officially a single character, and it occupies just one location in a character array
  - `'\0'` is a `char` with value 0 in C++

The STL String Class

Null-Terminated Strings

ASCII control characters

DEC	HEX	Simbolo ASCII	
00	00h	NULL	(carácter nulo)
01	01h	SOH	(inicio encabezado)
02	02h	STX	(inicio texto)
03	03h	ETX	(fin de texto)
04	04h	EOT	(fin transmisión)
05	05h	ENQ	(enquiry)
06	06h	ACK	(acknowledgement)
07	07h	BEL	(timbre)
08	08h	BS	(retroceso)
09	09h	HT	(tab horizontal)
10	0Ah	LF	(salto de linea)
11	0Bh	VT	(tab vertical)
12	0Ch	FF	(form feed)
13	0Dh	CR	(retorno de carro)
14	0Eh	SO	(shift Out)
15	0Fh	SI	(shift In)
16	10h	DLE	(data link escape)
17	11h	DC1	(device control 1)
18	12h	DC2	(device control 2)
19	13h	DC3	(device control 3)
20	14h	DC4	(device control 4)
21	15h	NAK	(negative acknowle.)
22	16h	SYN	(synchronous idle)
23	17h	ETB	(end of trans. block)
24	18h	CAN	(cancel)
25	19h	EM	(end of medium)
26	1Ah	SUB	(substitute)
27	1Bh	ESC	(escape)
28	1Ch	FS	(file separator)
29	1Dh	GS	(group separator)
30	1Eh	RS	(record separator)
31	1Fh	US	(unit separator)
127	20h	DEL	(delete)

ASCII printable characters

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
32	20h	espacio	64	40h	@	96	60h	`
33	21h	!	65	41h	A	97	61h	a
34	22h	"	66	42h	B	98	62h	b
35	23h	#	67	43h	C	99	63h	c
36	24h	$	68	44h	D	100	64h	d
37	25h	%	69	45h	E	101	65h	e
38	26h	&	70	46h	F	102	66h	f
39	27h	'	71	47h	G	103	67h	g
40	28h	(	72	48h	H	104	68h	h
41	29h	)	73	49h	I	105	69h	i
42	2Ah	*	74	4Ah	J	106	6Ah	j
43	2Bh	+	75	4Bh	K	107	6Bh	k
44	2Ch	,	76	4Ch	L	108	6Ch	l
45	2Dh	-	77	4Dh	M	109	6Dh	m
46	2Eh	.	78	4Eh	N	110	6Eh	n
47	2Fh	/	79	4Fh	O	111	6Fh	o
48	30h	0	80	50h	P	112	70h	p
49	31h	1	81	51h	Q	113	71h	q
50	32h	2	82	52h	R	114	72h	r
51	33h	3	83	53h	S	115	73h	s
52	34h	4	84	54h	T	116	74h	t
53	35h	5	85	55h	U	117	75h	u
54	36h	6	86	56h	V	118	76h	v
55	37h	7	87	57h	W	119	77h	w
56	38h	8	88	58h	X	120	78h	x
57	39h	9	89	59h	Y	121	79h	y
58	3Ah	:	90	5Ah	Z	122	7Ah	z
59	3Bh	;	91	5Bh	[	123	7Bh	{
60	3Ch	<	92	5Ch	\	124	7Ch	|
61	3Dh	=	93	5Dh	]	125	7Dh	}
62	3Eh	>	94	5Eh	^	126	7Eh	~
63	3Fh	?	95	5Fh	_			

theASCIIcode.com.ar

Extended ASCII characters

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
128	80h	Ç	160	A0h	á	192	C0h	└	224	E0h	Ó
129	81h	ü	161	A1h	í	193	C1h	┴	225	E1h	ß
130	82h	é	162	A2h	ó	194	C2h	┬	226	E2h	Ô
131	83h	â	163	A3h	ú	195	C3h	├	227	E3h	Ò
132	84h	ä	164	A4h	ñ	196	C4h	─	228	E4h	õ
133	85h	à	165	A5h	Ñ	197	C5h	┼	229	E5h	Õ
134	86h	å	166	A6h	ª	198	C6h	ã	230	E6h	µ
135	87h	ç	167	A7h	º	199	C7h	Ã	231	E7h	þ
136	88h	ê	168	A8h	¿	200	C8h	╚	232	E8h	Þ
137	89h	ë	169	A9h	®	201	C9h	╔	233	E9h	Ú
138	8Ah	è	170	AAh	¬	202	CAh	╩	234	EAh	Û
139	8Bh	ï	171	ABh	½	203	CBh	╦	235	EBh	Ù
140	8Ch	î	172	ACh	¼	204	CCh	╠	236	ECh	ý
141	8Dh	ì	173	ADh	¡	205	CDh	═	237	EDh	Ý
142	8Eh	Ä	174	AEh	«	206	CEh	╬	238	EEh	¯
143	8Fh	Å	175	AFh	»	207	CFh	¤	239	EFh	´
144	90h	É	176	B0h	░	208	D0h	ð	240	F0h	
145	91h	æ	177	B1h	▒	209	D1h	Ð	241	F1h	±
146	92h	Æ	178	B2h	▓	210	D2h	Ê	242	F2h	‗
147	93h	ô	179	B3h	│	211	D3h	Ë	243	F3h	¾
148	94h	ò	180	B4h	┤	212	D4h	È	244	F4h	¶
149	95h	ò	181	B5h	Á	213	D5h	ı	245	F5h	§
150	96h	û	182	B6h	Â	214	D6h	Í	246	F6h	÷
151	97h	ù	183	B7h	À	215	D7h	Î	247	F7h	¸
152	98h	ÿ	184	B8h	©	216	D8h	Ï	248	F8h	°
153	99h	Ö	185	B9h	╣	217	D9h	┘	249	F9h	¨
154	9Ah	Ü	186	BAh	║	218	DAh	┌	250	FAh	·
155	9Bh	ø	187	BBh	╗	219	DBh	█	251	FBh	¹
156	9Ch	£	188	BCh	╝	220	DCh	▄	252	FCh	³
157	9Dh	Ø	189	BDh	¢	221	DDh	¦	253	FDh	²
158	9Eh	×	190	BEh	¥	222	DEh	Ì	254	FEh	■
159	9Fh	ƒ	191	BFh	┐	223	DFh	▀	255	FFh	

Initializing a String Variable

- You can initialize a string variable when you declare it

❑ Example

```
char proclaim[20] = "Make it so.";
```

- Notice that the string assigned to the string variable need not fill the entire array

- **You can omit the array size** and C++ will automatically calculate the size to be exactly long enough to hold the string plus the null terminating character

❑ Example

```
char thought[ ] = "Peace";
```

What is the size of the `thought` array?

The Empty String

- String with no characters is called the **empty string**, and it is specified by two double quotes with nothing in between

❑ Example

```
char quiet[20] = "";
```

- There is not even a space between the two double quote marks

Reading and Writing String Variables

- C++ supports reading and writing string variables with the usual >> and << operators

❑ Example

```
char message[20] = "Noise";

cout << message;

cin >> message;
```

- The string-reading mechanism begins by skipping any **white space** in the input stream; **white space** consists of any blank, tab, or the return key
- The operation then reads characters until some more white space is encountered, placing these characters in the string variable
- The white space character itself is not read, **but a null terminating character is placed at the end of the string, making it a valid null-terminated string**

Using = and == with Strings

- Many of the usual operations simply do not work for strings
- **You cannot use a string variable in an assignment statement**
- **If you use == to test strings for equality, you will not get the result you expect**
  - Because: strings are implemented as arrays rather than simple values
- An attempt to assign a value to a string variable will quickly show the problem

valid

```
char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
char myword[] = "Hello";

cin >> myword;
```

invalid

```
myword = "Bye";
myword[] = "Bye";
```

valid

```
myword[0] = 'B';
myword[1] = 'y';
myword[2] = 'e';
myword[3] = '\0';
```

Using = and == with Strings (Cont'd)

❑Example

```
int main () {

    char greeting[10];
    greeting = "Hello";

    return 0;
}
```

```
In function 'int main()':
6:12: error: incompatible types in assignment of 'const char [6]' to
'char [10]'
```

- You can use the equals sign to assign a value to a string variable when the variable is declared
- The assignment operator throws an error if you try to assign a new value to the string

Using = and == with Strings (Cont'd)

- **You also cannot use the operator == in an expression to compare two strings for equality**
- **A good compiler will warn you that == actually tests to see whether the starting addresses of the arrays are the same,** but you will get incorrect results if you think you are testing for string equality
- For two string `string1` and `string2`, operator == always returns false
- There are ways around the string problems …

The STL String Class

Using = and == with Strings (Cont'd)

```
#include <iostream>

int main()
{
    char string1[] = "scu";
    char string2[] = "scu";

    if (string1 == string2)
         std::cout << "String are equal" << std::endl;
    else
         std::cout << "String are not equal" << std::endl;

    return 0;
}
```

```
Output:
String are not equal
Program ended with exit code: 0
```

The strcpy Function

- The easiest way to **assign a value** to a string variable is with the library function `strcpy` (from `<cstring>`)

❑ Example:

```
strcpy(greeting, "Hello");
```

```
char* strcpy(char target[ ], const char source[ ]);
// Precondition: source is a null-terminated string, and target is an
// array that is long enough to hold a copy of source.
// Postcondition: source has been copied to target, and the return
// value is a pointer to the first character of target.
```

- Notice that the **return value is a pointer** to a character, indicated by `char*` in the prototype, **this pointer points to the first character of the target array**

The strcpy Function (Cont'd)

```
#include <iostream>
int main()
{
    char string1[] = "scu";
    char string2[] = {'a', 'b', 'c'};
    strcpy(string1, string2);
    std::cout << string2 << std::endl;
    return 0;
}
```

```
Output:
abcabcscu
Program ended with exit code: 0
```

```
#include <iostream>
int main()
{
    char string1[] = "scu";
    char string2[] = {'a', 'b', 'c', '\0'};
    strcpy(string1, string2);
    std::cout << string2 << std::endl;
    return 0;
}
```

```
Output:
abc
Program ended with exit code: 0
```

The `strcat` Function

- The library function `strcat` (of the `cstring`) serves to add one string onto the end of another
- The "`cat`" in "`strcat`" comes from *catenate*
- The `strcat` function copies its second argument onto the end of its first argument

❑ Example

```
char greeting[20] = "Hello ";
strcat(greeting, "Good-bye");
```

"Good-bye" is added to the end of what's already in greeting

- After the function call, `greeting` contains "Hello Good-bye"

```
char* strcat(char target[ ], const char source[ ]);
// Precondition: target and source are null-terminated strings, and
// target is long enough to catenate source on the end.
// Postcondition: source has been catenated to target, and the
// return value is a pointer to the first character of target.
```

Dangers of `strcpy`, `strcat`, and Reading Strings

- Be careful using the `strcpy` and `strcat` functions, and also reading strings
- **None of these operations check that the string variable actually has sufficient room to hold the copied string**
- If you try to copy a string with 100 characters into an array of size 50:
  - You try to access an array beyond its declared bounds
  - Results in writing to memory locations that are not part of the array, often changing values of other declared variables
- During debugging, if you notice that a variable seems to be changing its value for no apparent reason, then think about the string variables and other arrays that your program uses, have you accessed a string variable or array beyond its declared size?

The `strlen` Function

- `strlen` (library function of `cstring`) **returns the number of characters in a null-terminated string**

```
size_t strlen(const char s[ ]);
// Precondition: s is a null-terminated string.
// Postcondition: The return value is the number of characters in s,
// up to (but not including) the null character.
```

❑ Example: `strlen("Hello Good-bye")` is 14

- The `strlen` function returns 0 for the length of the empty string
- **The null character is not counted**

The `strcmp` Function

- `strcmp` (library function of `cstring`) serves **to compare two strings**

```
int strcmp(const char s1[ ], const char s2[ ]);
// Precondition: s1 and s2 are null-terminated strings.
// Postcondition: The return value indicates the following:
// The return value is 0 -- s1 is equal to s2;
// The return value < 0 -- s1 is lexicographically before s2;
// The return value > 0 -- s1 is lexicographically after s2.
```

- `strcmp` returns zero if its two string arguments are equal to each other
- If the strings are not equal, then they are compared in the **lexicographic order**, which is the normal alphabetical order for ordinary words of all lower-case letters

❑ Example: `strcmp("chaos", "order")` will return some negative number, since `"chaos"` is alphabetically before `"order"`, while `strcmp("order", "chaos")` will return some positive integer

The String Class — Specification

- The **STL provides a string class that avoids the pitfalls of null-terminated strings**
- In particular, the string class **has a proper value semantics**
- Strings may also be compared using the usual six operators to test for equality (==), and various inequalities (`!=`, `>=`, `<=`, `>`, `<`)

Documentation for the Simple String Class

```
// FILE: mystring.h
// CLASS PROVIDED: string
// This is a simple version of the Standard Library string.
//
// CONSTRUCTOR for the string class:
// string(const char str[ ] = "") -- default argument is the empty
// string.
// Precondition: str is an ordinary null-terminated string.
// Postcondition: The string contains the sequence of chars from str.
//
// CONSTANT MEMBER FUNCTIONS for the string class:
// size_t length( ) const
// Postcondition: The return value is the number of characters in the
// string.
//
```

Documentation for the Simple String Class (Cont'd)

```
// char operator [ ](size_t position) const
// Precondition: position < length( ).
// Postcondition: The value returned is the character at the
// specified position of the string. A string's positions start from
// 0 at the start of the sequence and go up to length( )-1 at the
// right end.
//
// MODIFICATION MEMBER FUNCTIONS for the string class:
// void operator +=(const string& addend)
// Postcondition: addend has been catenated to the end of the string.
//
// void operator +=(const char addend[ ])
// Precondition: addend is an ordinary null-terminated string.
// Postcondition: addend has been catenated to the end of the string.
//
// void operator +=(char addend)
// Postcondition: The single character addend has been catenated to
// the end of the string.
```

Documentation for the Simple String Class (Cont'd)

```
// void reserve(size_t n)
// Postcondition: All functions will now work efficiently (without
// allocating new memory) until n characters are in the string.
//
// NON-MEMBER FUNCTIONS for the string class:
// string operator +(const string& s1, const string& s2)
// Postcondition: The string returned is the catenation of s1 and s2.
//
// istream& operator >>(istream& ins, string& target)
// Postcondition: A string has been read from the istream ins, and
// the istream ins is then returned by the function. The reading
// operation skips white space (i.e., blanks, newlines, tabs) at the
// start of ins.
// Then the string is read up to the next white space or the end of
// the file. The white space character that terminates the string has
// not been read.
//
// ostream& operator <<(ostream& outs, const string& source)
// Postcondition: The sequence of characters in source has been
// written to outs. The return value is the ostream outs.
```

The STL String Class

Documentation for the Simple String Class (Cont'd)

```
// void getline(istream& ins, string& target, char delimiter)
// Postcondition: A string has been read from the istream ins. The
// reading operation starts by skipping any white space, then reading
// all characters (including white space) until the delimiter is read
// and discarded (but not added to the end of the string). The return
// value is ins.
//
// VALUE SEMANTICS for the string class:
// Assignments and the copy constructor may be used with string
// objects.
//
// TOTAL ORDER SEMANTICS for the string class:
// The six comparison operators (==, !=, >=, <=, >, and <) are
// implemented for the string class, forming a total order semantics,
// using the usual lexicographic order on strings.
//
// DYNAMIC MEMORY usage by the string class:
// If there is insufficient dynamic memory then the following
// functions call new_handler: The constructors, resize, operator +=,
// operator +, and the assignment operator.
```

Constructor for the String Class

- The `string` class has a constructor with one argument:

```
string(const char str[ ] = "");
```

- The constructor initializes the string to contain the sequence of characters that is in the ordinary null-terminated string called `str`

❑ Example: If we want to create one of our strings that contains the sequence "`Peace`", then we may write:

```
char sequence[6] = "Peace";
string greeting(sequence);
```

- Without the variable `sequence`, we could also declare `greeting` as:

```
string greeting("Peace");
```

- Both approaches declare `greeting` to be one of our string objects that contains the sequence of characters "`Peace`"

The STL String Class

Overloading the operator [ ]

- One of the string's member functions is an overloaded operator with this specification:

```
char operator [ ](size_t position) const;
// Precondition: position < length( ).
// Postcondition: The value returned is the character at the
// specified position of the string. Note: A string's positions
// start from 0 at the star of the sequence and go up to length( )– 1
// at the right end.
```

- Allows you to use the syntax of square brackets to examine the individual characters of a string object

❑ Example

```
string greeting("Peace");
cout << greeting[0];    // Prints the P from greeting.
```

Some Further Overloading

- In our string class, there are three different += member functions with these prototypes:

```
void string::operator +=(const string& addend);
void string::operator +=(char addend);
void string::operator +=(const char addend[ ]);
```

- All three of these functions are called "`operator +=`" and all three can be used in a program
  - An example of overloading a single function name to carry out several related tasks
- When one of the functions is used, **the compiler looks at the type of the argument to determine which of the three functions to call**

Some Further Overloading (Cont'd)

- Example

```
string jack;
string adjective("nimble");

jack += adjective;
jack += '&';
jack += "quick";
```

Which member functions are called?

Other Operations for the String Class

- Our specification indicates that assignments and the copy constructor may be used with `string` objects (i.e., **a valid value semantics**)
- Since the string uses dynamic memory, you cannot rely on the automatic assignment operator and `copy` constructor
  - You must implement your own assignment operator and copy constructor
- There are also functions for reading, writing, and comparing strings

The String Class — Design

- With our design:
  - **A programmer can use strings with no worries about how long a string becomes**
- The plan is to have **a private member variable that is a dynamic array to hold the null-terminated string**
- Each member function ensures that the array has sufficient room, increasing the size of the array whenever necessary
- You can also explicitly set the size of the dynamic array that holds the null-terminated string, by calling the `reserve` function
  - Similar to the dynamic bag class, explicit resizing is not required
  - It is just a convenience for efficiency

The String Class — Design (Cont'd)

```
class string
{
    ...
 private:
    char *characters;
    std::size_t allocated;
    std::size_t current_length;
};
```

- The use of the member variables is controlled by **the invariant of the class**:
  - The string is stored as a null-terminated string in the dynamic array that `characters` points to
  - The total length of the dynamic array is stored in the member variable `allocated`
  - The total number of characters prior to the null character is stored in `current_length`, **which is always less than** `allocated`

The String Class — Implementation

- **Constructors.** The constructor is responsible for initializing the three private member variables
- The initialization occurs by copying a character sequence from an ordinary null-terminated string:

```
string::string(const char str[ ])
// Library facilities used: cstring
{
    current_length = strlen(str);
    allocated = current_length + 1;
    characters = new char[allocated];
    strcpy(characters, str);
}
```

- The constructor makes use of the library function `strcpy` to copy the null-terminated string from the parameter `str` to the dynamic array characters
- Make use of the library functions whenever they are needed

The String Class — Implementation (Cont'd)

- **The destructor.** Since the class uses dynamic memory, you must implement a destructor, your destructor will return the string's dynamic array to the heap

- **Comparison operators.** The string class has six comparison operators
- The prototype for the equality comparison is:

```
bool operator ==(const string& s1, const string& s2);
```

```
bool operator ==(const string& s1, const string& s2)
// Postcondition: The return value is true if s1 is identical to s2.
// Library facilities used: cstring
{
  return (strcmp(s1.characters, s2.characters) == 0);
}
```

Must be a friend, because it accesses private member variable `characters`

The String Class — Implementation (Cont'd)

- **The `reserve` function**
- Similar to the dynamic bag's reserve function

```
void reserve(size_t n);
// Postcondition: All functions will now work efficiently (without
// allocating new memory) until n characters are in the string.
```

- Programmers who use our string class never need to activate `reserve`, but it can be used to improve efficiency
- Our implementations of other member functions can also activate `reserve` whenever a larger array is needed

The String Class — Implementation (Cont'd)

- **The `operator >>` (the extraction operator)**
- After skipping the initial white space, our string input operator reads a string—reading up to but not including the next white space character (or until the input stream fails, which might occur from several causes, such as reaching the end of the file)
- The function `isspace` (from the `<cctype>` library facility) can help
  - This function has one argument (a character)
  - It returns true if its argument is one of the white space characters
- We can skip any initial white space with this loop:

```
while (ins && isspace(ins.peek( )))
      ins.ignore( );
```

The String Class — Implementation (Cont'd)

```
while (ins && isspace(ins.peek( )))
     ins.ignore( );
```

- The loop also uses three `istream` features:
  - In a boolean expression, the name of the `istream` (which is `ins`) acts as a test of whether the input stream is bad:
    - If `ins` returns a `true` value, then the stream is okay
    - A `false` value indicates a bad input stream
  - The `peek` member function returns the next character to be read (without actually reading it)
  - The `ignore` member function reads and discards the next character

The String Class — Implementation (Cont'd)

- After skipping the initial white space, your implementation should set the string to the empty string, and then read the input characters one at a time, adding each character to the end of the string
- The reading stops when you reach more white space (or the end of the file)
- Once the target string reaches its current capacity, our approach continues to work correctly
  - Although it is inefficient because target is probably resized by the += operator each time that we add another character
- Your documentation should warn programmers of this inefficiency so that a programmer can explicitly resize the target before calling the input operator
- An alternative method of reading input is provided by the `getline` function

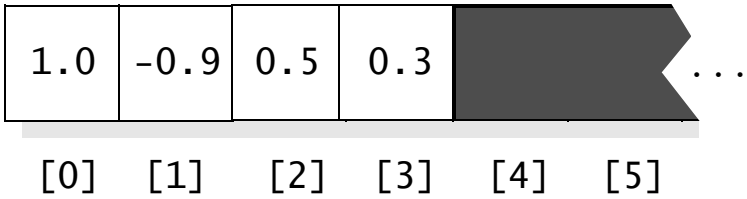
The Polynomial Class (second version) and a Project

Programming Project: The Polynomial

- **For this project, you should specify, design, and implement a class for polynomials**
- The coefficients are **double** numbers, and the exponents are **non-negative integers**
- The coefficients should be stored in a dynamic array of double numbers, with the exponent for the x^k term stored in location [k] of the array
- The maximum index of the array needs to be at least as big as the degree of the polynomial, so that the largest nonzero coefficient can be stored

❑ Example

$$0.3x^3 + 0.5x^2 - 0.9x + 1.0$$



- We also need:
  - A member variable to keep track of the current size of the dynamic array
  - A member variable to keep track of the current degree of the polynomial

Programming Project: The Polynomial

Constructors and destructor

```
polynomial( );              // Default constructor
polynomial(double a0);      // Set the x⁰ coefficient only
polynomial(const polynomial& source); // Copy constructor
~polynomial( );
```

- The default constructor creates a polynomial with all zero coefficients
- The second constructor creates a polynomial with the specified parameter as the coefficient of the x^0 term, and all other coefficients are zero

❑Example:

```
polynomial p(4.2); // p has only one nonzero term, 4.2
```

Programming Project: The Polynomial

Assignment operator

```
polynomial& operator = (const polynomial& source);
```

- The return type is `polynomial&` rather than `void`

❑ Example: If `a`, `b`, and `c` are three polynomials, we can write `a = b = c`

- Two facts for implementation:
  - The function implementation should return the object that activated the assignment, this is accomplished with the keyword `this`
    - `return *this;` means "return the object that `this` points to"
    - Since `this` always points to the object that activates the function, the return statement has the effect that we need
  - Using `polynomial&` as the return type permits a sequence of chained assignments
  - Can we use `this` inside a static member function?
  - Can we use `this` inside a friend function?

Programming Project: The Polynomial

Clarifying the this Keyword (Cont'd)

```
class Test
{

public:
    void printAddress () {cout << "My address is: " << this << endl;}
};

int main()
{
    Test obj1;

    obj1.printAddress();

    return 0;
}
```

```
Output:
My address is: 0x7fff5fbff6d8
```

Programming Project: The Polynomial

Clarifying the this Keyword (Cont'd)

```
class Test
{
private:
    int x, y;
public:
    Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
    Test& setX(int a) { x = a; return *this; }
    Test& setY(int b) { y = b; return *this; }
    void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
    Test obj1(5, 5);
    obj1.setX(10).setY(20);

    obj1.print();
    return 0;
}
```

- We return reference to the calling object
- This allows us to chain the operations

```
Output:
x = 10 y = 20
```

Programming Project: The Polynomial

Clarifying the this Keyword (Cont'd)

An alternative way – First approach

```
class Test
{
private:
    int x;
    int y;
public:
    Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
    Test* setX(int a) { x = a; return this; }
    Test* setY(int b) { y = b; return this; }
    void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
    Test obj1(5, 5);
    obj1.setX(10)->setY(20);

    obj1.print();
    return 0;
}
```

Output:

```
x = 10 y = 20
```

Programming Project: The Polynomial

Clarifying the `this` Keyword (Cont'd)

An alternative way – Second approach

```
class Test
{
private:
    int x;
    int y;
public:
    Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
    Test* setX(int a) { x = a; return this; }
    Test* setY(int b) { y = b; return this; }
    void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
    Test* obj1 = new Test(5, 5);
    obj1->setX(10)->setY(20);

    obj1->print();
    return 0;
}
```

Output:

```
x = 10 y = 20
```

Programming Project: The Polynomial

Arithmetic operators

- You can overload the binary arithmetic operators of addition, subtraction, and multiplication to add, subtract, and multiply two polynomials in the usual manner (Division is not possible, because it can result in fractional exponents)

❑ Example:

Suppose $q = 2x^3 + 4x^2 + 3x + 1$ and $r = 7x^2 + 6x + 5$.

Then: $q + r = 2x^3 + 11x^2 + 9x + 6$

$q - r = 2x^3 - 3x^2 - 3x - 4$

$q \times r = 14x^5 + 40x^4 + 55x^3 + 45x^2 + 21x + 5$

- The product, $q \times r$, is obtained by multiplying each separate term of q times each separate term of r and adding the results together

Summary

Chapter Summary

- **A pointer stores an address of another variable**
- Pointers are in particular used to point to **dynamically allocated memory**
- **Dynamic arrays** provide better flexibility since their size can vary according to need **during runtime**
- In C++, the **`new`** operator is used to **allocate dynamic memory**, and the **`delete`** operator is used to **free dynamic memory**
- The `new` operator usually indicates failure by throwing a special function exception `bad_alloc`
  - Normally the exception halts the program with an error message
- **Strings** and **bags** are two examples of classes that can be implemented with dynamic arrays

Chapter Summary

- **Classes that use dynamic memory should always include a `copy constructor`, an `overloaded assignment operator`, and a `destructor`**
  - The `copy constructor` and `assignment operator` must each copy an object by making a new copy of the dynamic memory (rather than just copying a pointer)
  - The destructor is responsible for freeing dynamic memory

Copyright Notice

The following copyright notices apply to some of the materials in this presentation: