Software Design and Analysis for Engineers

by

Dr. Lesley Shannon

Email: lshannon@ensc.sfu.ca

Course Website: http://www.ensc.sfu.ca/~lshannon/courses/ensc251

Simon Fraser University

Slide Set: 4

Date: September 21, 2015

What we're learning in this Slide Set:

- Arrays and Classes
- Classes, Dynamic memory, and Destructors
- Testing, Debugging and Verification

Textbook Chapters:

Relevant to this slide set:

- Sections 11.3-11.4
- Sections 5.4-5.5

Coming soon:

- Chapter 12 & 13

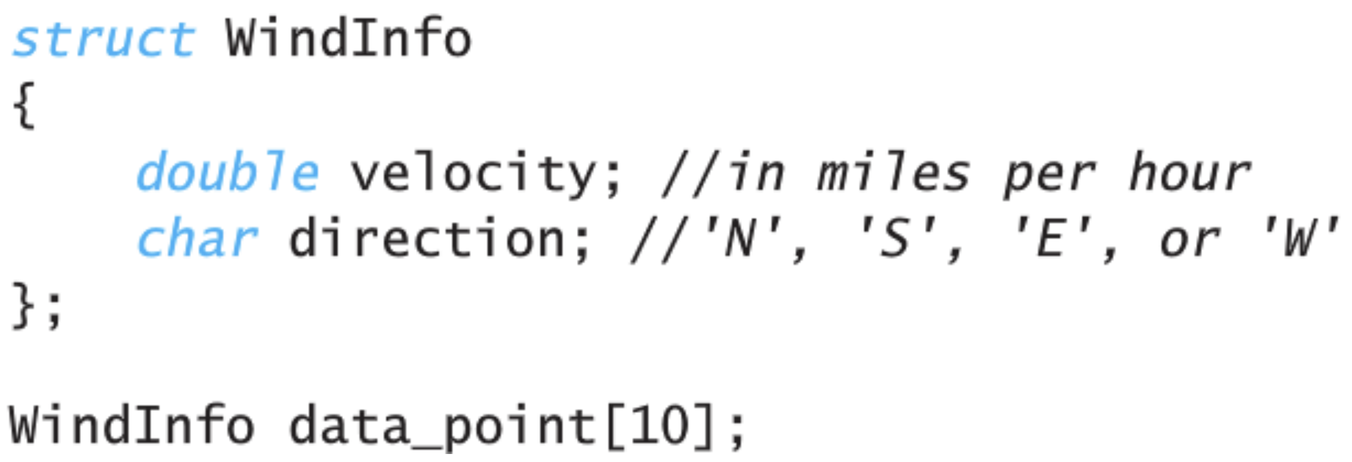
Arrays, Classes, and Structures:

You can combine arrays, structures and classes to create more complex data types:

- An Object that is an array of a class type,
- A structure with one field that is an array.
- A Class with a structure as a data member.
- Etc.

Arrays of Structures:

The base type of an array can be any type including user-defined types:



```
struct WindInfo
{
    double velocity; //in miles per hour
    char direction; //'N', 'S', 'E', or 'W'
};

WindInfo data_point[10];
```

This creates a variable, `data_point`, that is an array of 10 type WindInfo structures (indexed 0 to 9)

Remember that arrays have a statically defined size, so all of the space for the variable will be allocated when it is declared.

Arrays of Structures:

To access the data members of of each array member of `data_point`

```
int i;
for (i = 0; i < 10; i++)
{
    cout << "Enter velocity for "
         << i << " numbered data point: ";
    cin >> data_point[i].velocity;
    cout << "Enter direction for that data point"
         << " (N, S, E, or W): ";
    cin >> data_point[i].direction;
}
```

Note that you index the array first and then use the dot operator to indicate which data member within that data point you wish to access:

`data_point[i].velocity` or `data_point[i].direction`

Arrays of Classess:

Declaring an array of a base class type is basically the same:

```
Money difference[5];
```

NOTE: Creating an Object that is an array of a class type calls the Default Constructor to initialize each of the indexed variables.

Otherwise arrays of class type/structures/defined types can be treated basically the same.

Let’s see some example code:

```
class Money
{
public:
friend Money operator +(const Money& amount1, const Money& amount2);
//Returns the sum of the values of amount1 and amount2.

friend Money operator -(const Money& amount1, const Money& amount2);
//Returns amount1 minus amount2.

friend Money operator -(const Money& amount);
//Returns the negative of the value of amount.

friend bool operator ==(const Money& amount1, const Money& amount2);
//Returns true if amount1 and amount2 have the same value; false otherwise.

friend bool operator <(const Money& amount1, const Money& amount2);
//Returns true if amount1 is less than amount2; false otherwise.
Money(long dollars, int cents);
//Initializes the object so its value represents an amount with
//the dollars and cents given by the arguments. If the amount
//is negative, then both dollars and cents should be negative.

Money(long dollars);
//Initializes the object so its value represents $dollars.00.

Money( );
//Initializes the object so its value represents $0.00.
```

Remainder of `Money` class definition:

```
double get_value( ) const;
//Returns the amount of money recorded in the data portion of the calling
//object.

friend istream& operator >>(istream& ins, Money& amount);
//Overloads the >> operator so it can be used to input values of type
//Money. Notation for inputting negative amounts is as in - $100.00.
//Precondition: If ins is a file input stream, then ins has already been
//connected to a file.

friend ostream& operator <<(ostream& outs, const Money& amount);
//Overloads the << operator so it can be used to output values of type
//Money. Precedes each output value of type Money with a dollar sign.
//Precondition: If outs is a file output stream, then outs has already been
//connected to a file.
private:
    long all_cents;
};
```

main:

```
//Reads in 5 amounts of money and shows how much each
//amount differs from the largest amount.

int main( )
{
    Money amount[5], max;
    int i;

    cout << "Enter 5 amounts of money:\n";
    cin >> amount[0];
    max = amount[0];
    for (i = 1; i < 5; i++)
    {
        cin >> amount[i];
        if (max < amount[i])
            max = amount[i];
        //max is the largest of amount[0], . . ., amount[i].
    }

    Money difference[5];
    for (i = 0; i < 5; i++)
        difference[i] = max - amount[i];

    cout << "The highest amount is " << max << endl;
    cout << "The amounts and their\n"
         << "differences from the largest are:\n";
    for (i = 0; i < 5; i++)
    {
        cout << amount[i] << " off by "
             << difference[i] << endl;
    }

    return 0;
}
```

How to use arrays as class/structure members:

First, here's a structure with a field that is an array -

```
struct Data
{
    double time[10];
    int distance;
};

Data my_best;
```

You can then initialize the data fields (member variables) using:

```
my_best.distance = 20;
```

and:

```
cout << "Enter ten times (in seconds):\n";
for (int i = 0; i < 10; i++)
    cin >> my_best.time[i];
```

Here you use the dot operator first and then index the field

Here's a class definition with a data member that is an array:

```
const int MAX_LIST_SIZE = 50;

class TemperatureList
{
public:
    TemperatureList( );
    //Initializes the object to an empty list.

    void add_temperature(double temperature);
    //Precondition: The list is not full.
    //Postcondition: The temperature has been added to the list.

    bool full( ) const;
    //Returns true if the list is full; false otherwise.

    friend ostream& operator <<(ostream& outs,
        const TemperatureList& the_object);
    //Overloads the << operator so it can be used to output values of
    //type TemperatureList. Temperatures are output one per line.
    //Precondition: If outs is a file output stream, then outs
    //has already been connected to a file.

private:
    double list[MAX_LIST_SIZE]; //of temperatures in Fahrenheit
    int size; //number of array positions filled
};
```

The member function implementations:

```
TemperatureList::TemperatureList( ) : size(0)
{
    //Body intentionally empty.
}
void TemperatureList::add_temperature(double temperature)
{//Uses iostream and cstdlib:
    if ( full( ) )
    {
        cout << "Error: adding to a full list.\n";
        exit(1);
    }
    else
    {
        list[size] = temperature;
        size = size + 1;
    }
}

bool TemperatureList::full( ) const
{
    return (size == MAX_LIST_SIZE);
}
```

The member function implementations cont'd & assignments:

```
//Uses iostream:
ostream& operator <<(ostream& outs, const TemperatureList& the_object)
{
    for (int i = 0; i < the_object.size; i++)
        outs << the_object.list[i] << " F\n";
    return outs;
}
```

You would declare an object of type TemperatureList class as follows:

```
TemperatureList my_data;
```

You can store an additional temperature to the list by:

```
my_data.add_temperature(77);
```

Or output the list: `cout << my_data;`

You can also check if the list is full.

However, you cannot delete a single temperature; all you can do is initialize the entire list to empty: `my_data = TemperatureList();`

Dynamic Arrays, Classes, and Structures:

You can combine dynamic arrays, structures and classes to create more complex data types (that are more memory efficient:

- An Object that is a dynamic array of a class type,
- A structure with one field that is an dynamic array/pointer.
- Etc.

Remember: if you are going to allocate ("create") new memory dynamically in your program, you must "destroy" it (return it to the heap when you are done:

- This is why we have destructors

Let's see an example

Constructors and the Destructor for the StringVar class

```
class StringVar
{
public:
    StringVar(int size);
    //Initializes the object so it can accept string values up to size
    //in length. Sets the value of the object equal to the empty string.

    StringVar( );
    //Initializes the object so it can accept string values of length 100
    //or less. Sets the value of the object equal to the empty string.

    StringVar(const char a[]);
    //Precondition: The array a contains characters terminated with '\0'.
    //Initializes the object so its value is the string stored in a and
    //so that it can later be set to string values up to strlen(a) in length.

    StringVar(const StringVar& string_object);
    //Copy constructor.

    ~StringVar( );
    //Returns all the dynamic memory used by the object to the freestore.
```

StringVar class definition (cont'd)

```
  int length( ) const;
  //Returns the length of the current string value.

  void input_line(istream& ins);
  //Precondition: If ins is a file input stream, then ins has been
  //connected to a file.
  //Action: The next text in the input stream ins, up to '\n', is copied
  //to the calling object. If there is not sufficient room, then
  //only as much as will fit is copied.

  friend ostream& operator <<(ostream& outs, const StringVar& the_string);
  //Overloads the << operator so it can be used to output values
  //of type StringVar
  //Precondition: If outs is a file output stream, then outs
  //has already been connected to a file.
private:
    char *value; //pointer to dynamic array that holds the string value.
    int max_length; //declared max length of any string value.
};
```

An example main program using the non-member function `conversation()`:

```
void conversation(int max_name_size);
//Carries on a conversation with the user.

int main( )
{
    using namespace std;
    conversation(30);
    cout << "End of demonstration.\n";
    return 0;
}

//This is only a demonstration function:
void conversation(int max_name_size)
{
    using namespace std;

    StringVar your_name(max_name_size), our_name("Borg");

    cout << "What is your name?\n";
    your_name.input_line(cin);
    cout << "We are " << our_name << endl;
    cout << "We will meet again " << your_name << endl;
}
```

Memory is returned to the freestore when the function call ends.

Determines the size of the dynamic array

Let’s look at how to implement the member functions for this class, starting with the normal member functions.

“Normal” Member functions:

```
 //Uses cstring:
 int StringVar::length( ) const
 {
     return strlen(value);
 }

 //Uses iostream:
 void StringVar::input_line(istream& ins)
 {
     ins.getline(value, max_length + 1);
 }

 //Uses iostream:
ostream& operator <<(ostream& outs, const StringVar& the_string)
 {
     outs << the_string.value;
     return outs;
 }
```

StringVar constructors:

```
//Uses cstddef and cstdlib:
StringVar::StringVar(int size) : max_length(size)
{
    value = new char[max_length + 1];//+1 is for '\0'.
    value[0] = '\0';
}

//Uses cstddef and cstdlib:
StringVar::StringVar( ) : max_length(100)
{
    value = new char[max_length + 1];//+1 is for '\0'.
    value[0] = '\0';
}

//Uses cstring, cstddef, and cstdlib:
StringVar::StringVar(const char a[]) : max_length(strlen(a))
{
    value = new char[max_length + 1];//+1 is for '\0'.
    strcpy(value, a);
}
```

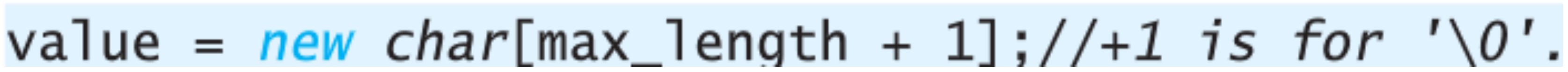
Recall the constructor call in `conversation()`

```
StringVar your_name(max_name_size), our_name("Borg");
```

Let’s review what the “new” operator does:

`new` creates a new dynamic variable of a specified type and returns a pointer that points to the starting address of the memory allocated to this variable.

For example, since value is a char pointer, to dynamically create an array we call:



```
value = new char[max_length + 1];//+1 is for '\0'.
```

What is the ‘\0’ character for?

Challenges of Dynamic memory:

When you create dynamic memory, you must almost “destroy” or `delete` it

Even if the dynamic variable is created for a local pointer in a function call, when the local pointer goes away at the end of the function, the dynamically allocated memory will not be destroyed (returned to the free-store) unless you call delete

-Why do you think this is?

When you do not destroy/delete memory you have dynamically allocated, you create a memory leak. If this is repeated often enough, the system will run out of available memory in the free-store and crash/abort to protect the application from overwriting the O/S and other protected memory

WARNING: Never mix C++'s “new/delete” with C's “malloc/free.” They use different portions of the dynamic memory segment (free-store versus heap). Although they are conceptually rather similar, they are not the same.

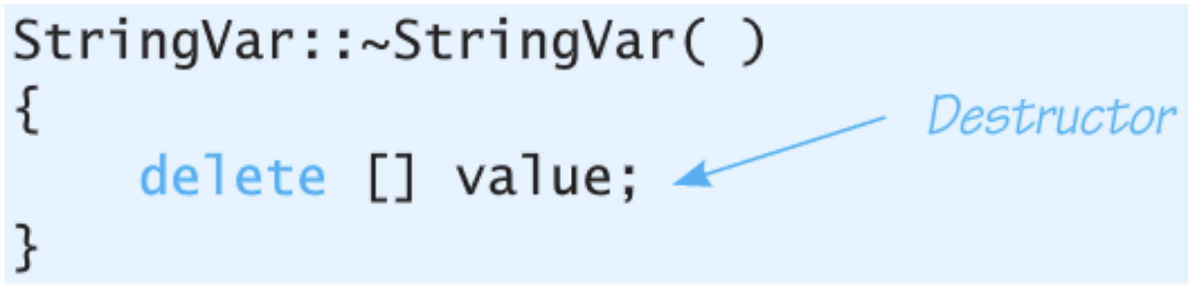
We won't go into the finer details of the two different abstractions, but can you think why mixing calls (new & free; malloc & delete) will cause problems?

All these considerations are particularly dangerous when you create a class to be used by someone else.

If you allocate memory inside of a class implementation, the programmer won’t know about it and won’t delete it.

- Worse, if you create your data members as private members, the programmer cannot generally access the needed pointer variables so they cannot call delete with these pointer variables
  - This doesn’t mean you should make your variables public; this means that it is your responsibility to make sure you responsibly create and destroy member.

To handle this problem, C++ has a specific member function called a Destructor:



```
StringVar::~StringVar( )
{
    delete [] value;
}
```

First, let's recall what the delete operator does.

The “delete” operator:

Destroys the dynamic variable and returns the memory that it occupied back to the free-store.

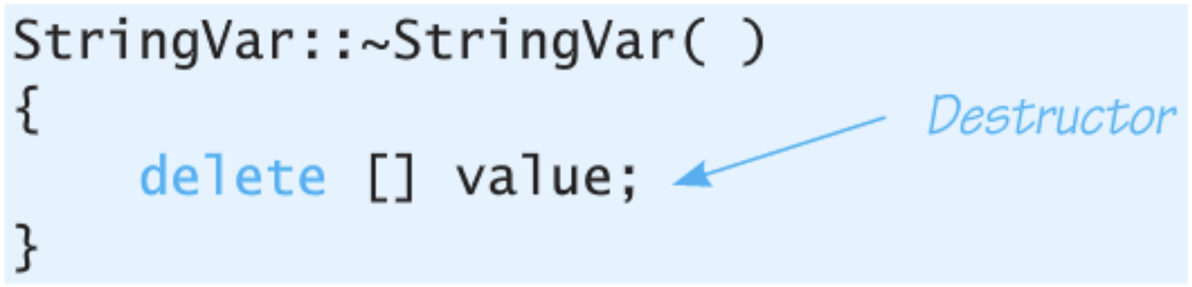
- The returned memory can then be reused when a `new` dynamic variable is allocated.

`delete` takes a pointer to an object as a parameter.

If you want to delete an array of objects, you need to include [].

- The [] ensures that the delete function first determines the length of the array of the objects and then deletes the entire array of objects.
- If you don’t include the [], it will compile but the behaviour will likely not be what you want.
- It will depend on the compiler, but you likely won’t delete the array.

To handle this problem, C++ has a specific member function called a Destructor:



```
StringVar::~StringVar( )
{
    delete [] value;
}
```

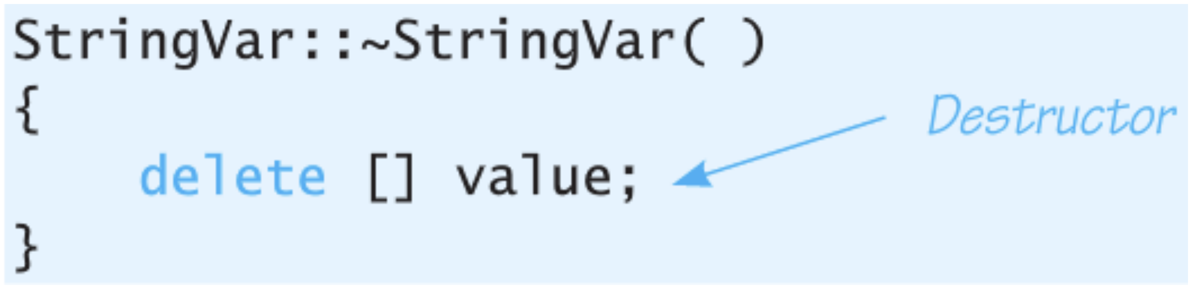
A destructor is a class member function that is called ***<u>automatically</u>*** when an object of the class passes out of scope

For example: This applies to local variables in in a function call. If you have a local variable in a function that has a destructor, then when the function terminates, the destructor for that object is automatically called.

As long as the destructor is defined correctly, it should delete all of the dynamic variables in the object.

- Depending on the structure of the data members in the object, this may require one or more calls of delete
- You may also use it to perform other operations to “clean up,” but returning the dynamic memory to the free-store is the key requirement.

StringVar's Destructor:

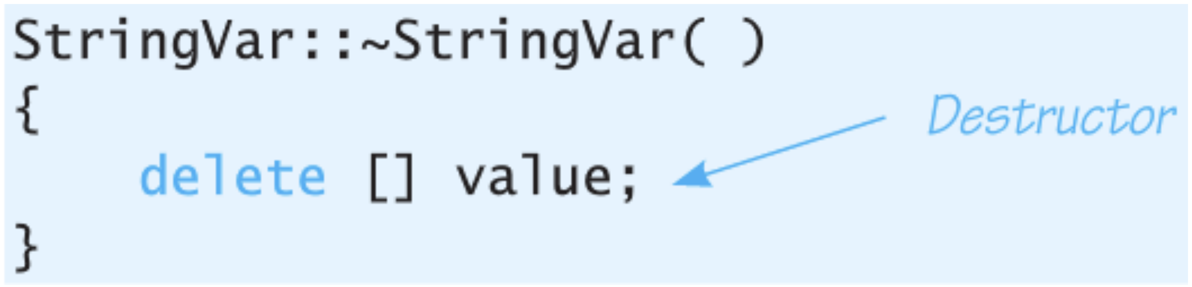


```
StringVar::~StringVar( )
{
    delete [] value;
}
```

Like a Constructor:

- Destructors always have the same name as their class but are preceded by a tilde symbol ('~')
- Destructors have no type for the returned value (not even void).

StringVar’s Destructor:



```
StringVar::~StringVar( )
{
    delete [] value;
}
```

The main differences from a Constructor are:

- Destructors always have no parameters (like a default Constructor)
- You can only have one Destructor per class (you cannot overload the destructor for a class).

Recall the function conversation():

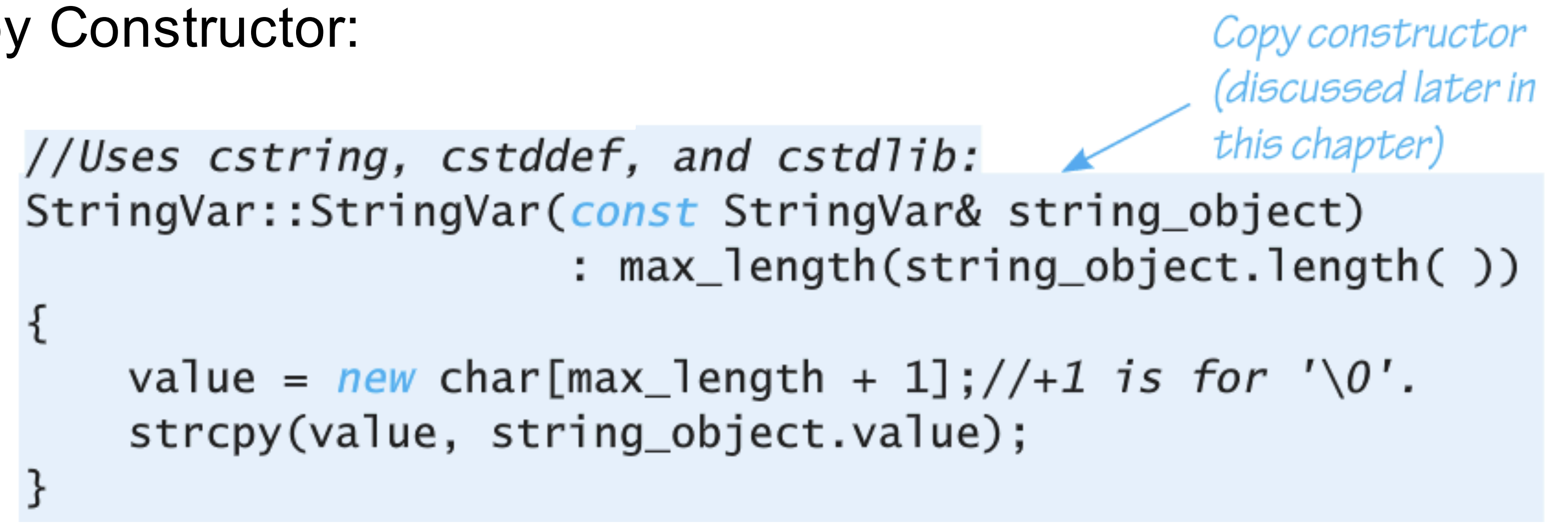
```
//This is only a demonstration function:
void conversation(int max_name_size)
{
    using namespace std;

    StringVar your_name(max_name_size), our_name("Borg");

    cout << "What is your name?\n";
    your_name.input_line(cin);
    cout << "We are " << our_name << endl;
    cout << "We will meet again " << your_name << endl;
}
```

Determines the size of the dynamic array

Without the destructor, at the end of the function, the `your_name` and `our_name` objects would be destroyed, but the char arrays that are pointed to the data member `*value`, would not.

In the case of this example, the program ends soon after, so it won’t crash. But if you repeated this behaviour without a destructor, you would consume all of the free-store memory eventually and crash your application.

Copy Constructor:



Copy constructor (discussed later in this chapter)

```
//Uses cstring, cstddef, and cstdlib:
StringVar::StringVar(const StringVar& string_object)
                    : max_length(string_object.length( ))
{
    value = new char[max_length + 1];//+1 is for '\0'.
    strcpy(value, string_object.value);
}
```

A copy constructor is a constructor that has one parameter of the same type as the class itself.

It must be a call by reference parameter

Normally it is the parameter is preceded by the const parameter modifier (so it is a constant parameter even though it is called by reference).

Otherwise Copy Constructors are the same as other constructors

Recall the Constructors and the Destructor for the StringVar class

```
class StringVar
{
public:
    StringVar(int size);
    //Initializes the object so it can accept string values up to size
    //in length. Sets the value of the object equal to the empty string.

    StringVar( );
    //Initializes the object so it can accept string values of length 100
    //or less. Sets the value of the object equal to the empty string.

    StringVar(const char a[]);
    //Precondition: The array a contains characters terminated with '\0'.
    //Initializes the object so its value is the string stored in a and
    //so that it can later be set to string values up to strlen(a) in length.

    StringVar(const StringVar& string_object);
    //Copy constructor.

    ~StringVar( );
    //Returns all the dynamic memory used by the object to the freestore.
```

Copy Constructor:

Copy constructor (discussed later in this chapter)

```
//Uses cstring, cstddef, and cstdlib:
StringVar::StringVar(const StringVar& string_object)
                    : max_length(string_object.length( ))
{
    value = new char[max_length + 1];//+1 is for '\0'.
    strcpy(value, string_object.value);
}
```

For example:

```
StringVar line(20), motto("Constructors can help.");
cout << "Enter a string of length 20 or less:\n";
line.input_line(cin);
StringVar temp(line);//Initialized by the copy constructor.
```

`temp` is initialized using the copy constructor

We want it to be an <u>independent</u> object with a complete copy of the data members in `line`

We do not want `temp.value` to point to the same place as `line.value`

Copy Constructor:

Copy constructor (discussed later in this chapter)

```
//Uses cstring, cstddef, and cstdlib:
StringVar::StringVar(const StringVar& string_object)
                           : max_length(string_object.length( ))
{
     value = new char[max_length + 1];//+1 is for '\0'.
     strcpy(value, string_object.value);
}
```

For example:

```
StringVar line(20), motto("Constructors can help.");
cout << "Enter a string of length 20 or less:\n";
line.input_line(cin);
StringVar temp(line);//Initialized by the copy constructor.
```

Since we do not want `temp.value` to point to the same place as `line.value`

- The copy constructor creates a new array for value and then copies the contents of original array to the new one.

While this "looks" nice, do you understand why we need the copy constructor?

Some reminders about pointers:

```
int *p; //Declares a pointer to an int

int v; // Declares a variable of type int

p = &v; //Assigns p the address of v

*p = 44; // Dereferences the pointer p to assign 44 to
the variable v

void foo (int *value); //The input parameter is an int
                       //pointer-> passed as a call by
                       //value; must be dereferenced to
                       //manipulate the int value

void bar (int &value); //Call by reference, passes the
                       //address of the int to the
                 //function, but you don’t need
                 //to dereference value to access
                 //the int stored in the variable.
```

Look at what happens when you pass a pointer to a function

```
//Program to demonstrate the way call-by-value parameters
//behave with pointer arguments.
#include <iostream>
using namespace std;

typedef int* IntPointer;

void sneaky(IntPointer temp);

int main( )
{
    IntPointer p;

    p = new int;
    *p = 77;
    cout << "Before call to function *p == "
         << *p << endl;

    sneaky(p);

    cout << "After call to function *p == "
         << *p << endl;

    return 0;
}

void sneaky(IntPointer temp)
{
    *temp = 99;
    cout << "Inside function call *temp == "
         << *temp << endl;
}
```

What is the value of *p (the integer p points to)?

Why?

If a function has an input parameter type that is a class or structure type with member variables of a pointer type, the same thing can occur even with call by value arguments of the class type.

This is an example where you might want to use a copy constructor.

For example, what would happen in the following example if we did not include a copy constructor in the definition of class StringVar but included the following function (note its input parameter is call by value)?

With this function definition:

```
void show_string(StringVar the_string)
{
    cout << "The string is: "
         << the_string << endl;
}
```

and this piece of code, what happens:

```
StringVar greeting("Hello");
show_string(greeting);
cout << "After call: " << greeting << endl;
```

Line 1: The object greeting is created, with value pointing to "Hello".

Line 2: When the function `show_string` is called, a copy of the object `greeting` is passed into `show_string`

WARNING: This means that the value of `the_string.value` is set equal to `greeting.value`

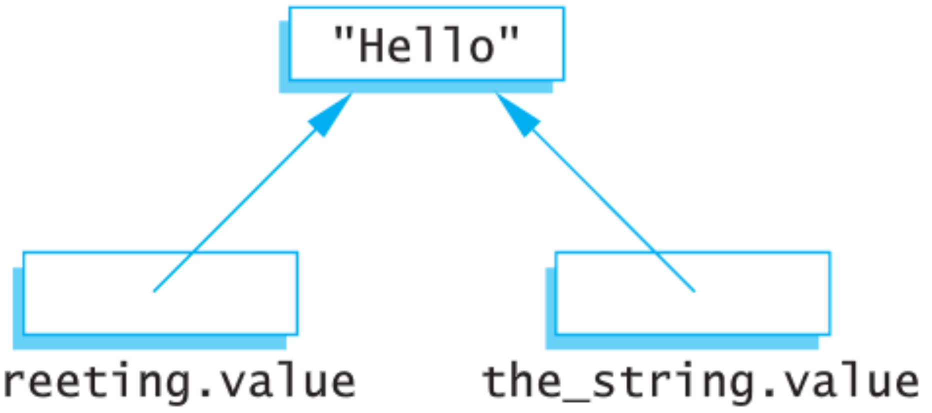
With this function definition:

```
void show_string(StringVar the_string)
{
    cout << "The string is: "
         << the_string << endl;
}
```

and this piece of code, what happens:

```
StringVar greeting("Hello");
show_string(greeting);
cout << "After call: " << greeting << endl;
```

Since `the_string.value` is set equal to `greeting.value`, we have the following:



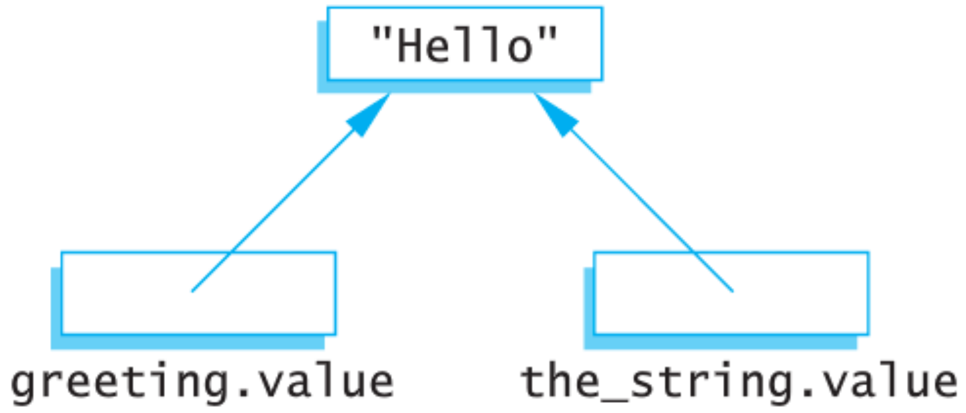
With this function definition:

```
void show_string(StringVar the_string)
{
    cout << "The string is: "
         << the_string << endl;
}
```

and this piece of code, what happens:

```
StringVar greeting("Hello");
show_string(greeting);
cout << "After call: " << greeting << endl;
```

When `show_string()` finishes, the destructor is called to return the memory dynamically allocated to `the_string` to free-store:



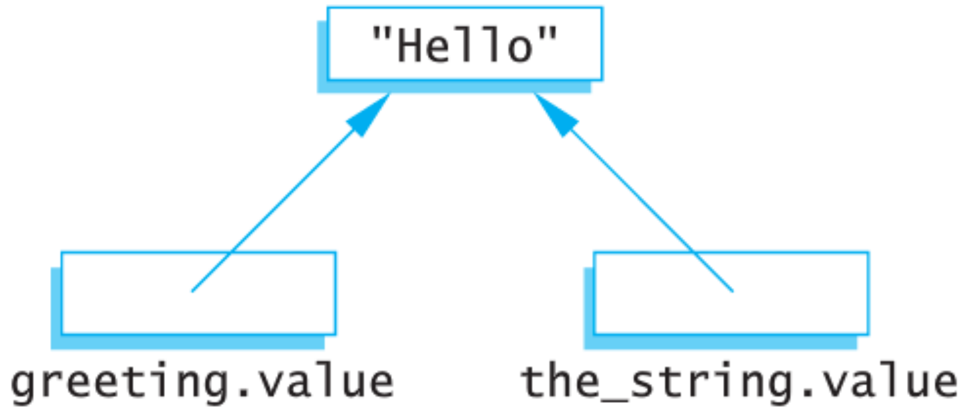
With this function definition:

```
void show_string(StringVar the_string)
{
    cout << "The string is: "
         << the_string << endl;
}
```

and this piece of code, what happens:

```
StringVar greeting("Hello");
show_string(greeting);
cout << "After call: " << greeting << endl;
```

Since `delete [] value;` is applied to the object `the_string`, we have `delete [] the_string.value;`



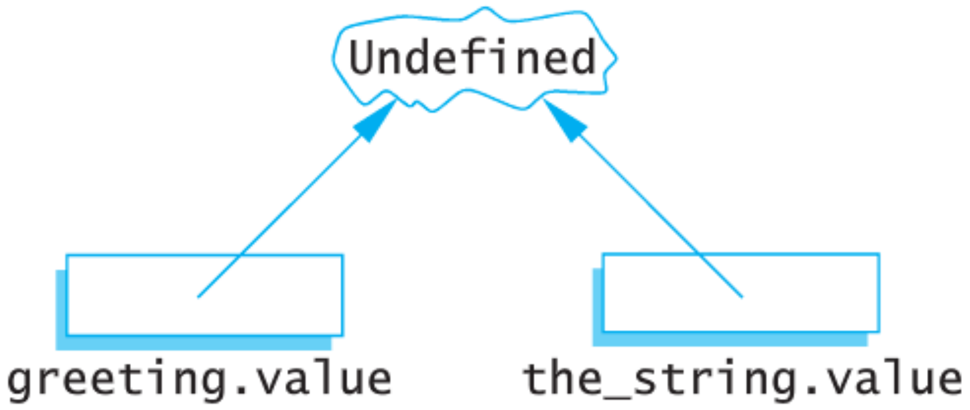
With this function definition:

```
void show_string(StringVar the_string)
{
    cout << "The string is: "
         << the_string << endl;
}
```

and this piece of code, what happens:

```
StringVar greeting("Hello");
show_string(greeting);
cout << "After call: " << greeting << endl;
```

Since `the_string.value` is equal to `greeting.value`, we have:



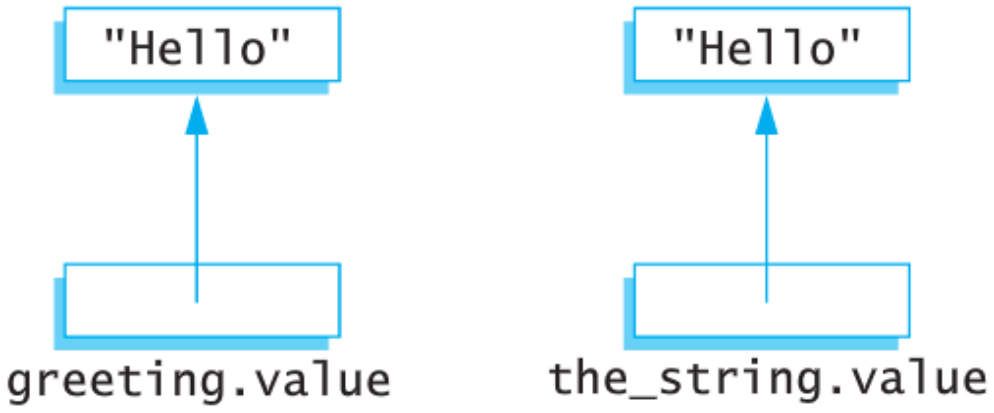
This will cause a problem in Line 3 of our code above.

What is the value of greeting given to cout?

There will be additional problems later when we try and destroy the object greeting. Since greeting.value has already been deleted once, you cannot delete it again.

- Trying to do so can produce a system error that will crash your program.

By including a copy constructor, none of this happens. Instead, the copy constructor creates the new “the_string” object, and we have the following:



This means any changes made to `the_string` have no impact on `greeting` and there are no problems with the destructor.

The same thing applies to a function that returns a value of a class type.

- The copy constructor is automatically called to copy the value specified by the return statement.
  - If there is no copy constructor, you can end up with a similar problem to what we just described for call by value parameters above

General rule:

You must include a Copy Constructor in your class definition if a class definition utilizes pointers and dynamically allocated memory using the new operator.

If a class does not involve dynamically allocated memory/pointers, you don’t need a copy constructor.

Also note:

The copy constructor is not called when you use the assignment operator (and set one object equal to another).

- Copy Constructors may be used for initialization to create a new object.
- The Assignment operator modifies an existing object so that it has an identical copy (in its own location) of the RHS argument.

Summary:

- Copy constructors are automatically called when function returns a value of the class type or when a class type is used as a call-by-parameter value.
- Copy constructors can also be used like normal constructors for normal initialization.
- Any class that has dynamic memory allocation (i.e. pointers and the new operator) should have a copy constructor.

Keep in mind that if you need to create a copy constructor and destructor, then you likely need to overload the “=“ operator.

- The default copy constructors and overloaded “=“ operator generated by the compiler will work fine for predefined types (int, float, etc.).
- These defaults are likely to misbehave with pointers and the new operator.

So the safest way is to create your own copy constructor, overloaded “=“ operator, and destructor

Overloading the Assignment Operator:

Suppose we have the following:

```
StringVar string1(10), string2(20);
```

and

```
string1 = string2;
```

Remember, the default overloaded assignment operator copies the values of the member variables of `string2` to `string1`.

This is fine when copying the `max_length` fields, but it is a problem when copying the `value` fields as now both objects now point to the same string.

This means that any change of the contents pointed to by the `value` field of one object effects both objects.

To fix this we need to overload the '=' operator.

Overloading the Assignment Operator:

You cannot overload the assignment operator in the same fashion as the other overloaded operators we looked at (‘+’, “<<“ etc.).

In this case, the overloaded assignment operator must be a member of the class and not a friend function of the class as we have done previously.

For example:

```
class StringVar
{
public:
    void operator =(const StringVar& right_side);
    //Overloads the assignment operator = to copy a string
    //from one object to another.
    <The rest of the definition of the class can be the same as in
     Display 11.11.>
```

Using this declaration, the assignment operator can be used as normal:

```
string1 = string2;
```
//(string 1 is the calling object and string2 is the
// argument).

Overloading the Assignment Operator:

The following is an example definition of the overloaded assignment operator:

```
//The following is acceptable, but
//we will give a better definition:
void StringVar::operator =(const StringVar& right_side)
{
    int new_length = strlen(right_side.value);
    if ((new_length) > max_length)
        new_length = max_length;

    for (int i = 0; i < new_length; i++)
        value[i] = right_side.value[i];
    value[new_length] = '\0';
}
```

Unfortunately, if your new string is longer, the extra characters get chopped off. To solve this problem, you could try…

Overloading the Assignment Operator:

If we try:

```
//This version has a bug:
void StringVar::operator =(const StringVar& right_side)
{
    delete [] value;
    int new_length = strlen(right_side.value);
    max_length = new_length;

    value = new char[max_length + 1];

    for (int i = 0; i < new_length; i++)
        value[i] = right_side.value[i];
    value[new_length] = '\0';
}
```

What is the problem?

Overloading the Assignment Operator:

If we try:

```
//This version has a bug:
void StringVar::operator =(const StringVar& right_side)
{
    delete [] value;
    int new_length = strlen(right_side.value);
    max_length = new_length;

    value = new char[max_length + 1];

    for (int i = 0; i < new_length; i++)
        value[i] = right_side.value[i];
    value[new_length] = '\0';
}
```

What happens if you: `my_string = my_string;`

The first statement is: `delete [] value;`

… which is basically: `delete [] my_string.value;`

meaning that your pointer is undefined.

Overloading the Assignment Operator:

This is better:

```
//This is our final version:
void StringVar::operator =(const StringVar& right_side)
{
    int new_length = strlen(right_side.value);
    if (new_length > max_length)
    {
        delete [] value;
        max_length = new_length;
        value = new char[max_length + 1];
    }
    for (int i = 0; i < new_length; i++)
        value[i] = right_side.value[i];
    value[new_length] = '\0';
}
```

In this case, you only delete and create a new char array if the current one is not large enough.

Warning: Assigning an object to itself is a special case that you should always consider when overloading the assignment operator.

In many cases the obvious definition for overloading the assignment operator will not work for this case.

You should always perform a test for this special case and write your function definition to work in <u>all</u> cases.

And now for some practical talk on Commenting, Testing, Debugging, and Designing for Testability

Recall: Commenting is to provide the user with more information; to improve the usability and readability of your code.

In Section 5.3, the concept of Precondition and Post-conditions is discussed.

This is a useful part of your standard interface when defining a class:

Remember comments are part of the standard interface and should tell the user how to use your public member functions.

Specifying preconditions (what you expect as input) and post-conditions (what you will create as an output) are extremely valuable for this.

However, you don’t want to make this description dependent on the types of the member variables.

For example:

```
void post_interest(double& balance, double rate);
//Precondition: balance is a nonnegative savings
//account balance.rate is the interest rate
//expressed as a percent, such as 5 for 5%.
//Postcondition: The value of balance has been
//increased by rate percent.
```

You don’t need to see the function body to understand how to use the function.

For example:

```
void post_interest(double& balance, double rate);
//Precondition: balance is a nonnegative savings
//account balance.rate is the interest rate
//expressed as a percent, such as 5 for 5%.
//Postcondition: The value of balance has been
//increased by rate percent.
```

Creating the standard interface first is a good way to start the design process.

- It allows team members to work in parallel
- It also helps you identify all of the test cases that need to be covered.

Agile Software design relies on Unit Testing:

The designer creates a set of test functions for each module (function) in the design.

These tests are created FIRST.

- You then write only sufficient code to pass the tests and no more.
- The result should be that your tests may be a bit clunky, but your actual program code should be elegant and clean

When creating your unit tests, you need to cover:

-the “normal” cases,

-the boundary/corner cases

-illegal cases

Every function should be tested independently and then in the context of the program it is being used in

**Preferably every other function in this program should already have been tested

Agile Software design relies on Unit Testing:

To test a function outside of the program you want to use it in, you need to use a Driver Program or Testbed.

These programs are temporary tools that allow you to activate the different portions of a function to verify their functionality independent of the actual program.

- If you can, it is good to be able to make them reusable

To test the get_input function with the standard interface:

```
void get_input(double& cost, int& turnover);
//Precondition: User is ready to enter values correctly.
//Postcondition: The value of cost has been set to the
//wholesale cost of one item. The value of turnover has been
//set to the expected number of days until the item is sold.
```

and the implementation:

```
//Uses iostream:
void get_input(double& cost, int& turnover)
{
    using namespace std;
    cout << "Enter the wholesale cost of item: $";
    cin >> cost;
    cout << "Enter the expected number of days until sold: ";
    cin >> turnover;
}
```

your driver program could be… (see next slide):

Driver program for the function get_input:

```
int main( )
{
    using namespace std;
    double wholesale_cost;
    int shelf_time;
    char ans;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    do
    {
        get_input(wholesale_cost, shelf_time);

        cout << "Wholesale cost is now $"
             << wholesale_cost << endl;
        cout << "Days until sold is now "
             << shelf_time << endl;

        cout << "Test again?"
             << " (Type y for yes or n for no): ";
        cin >> ans;
        cout << endl;
    } while (ans == 'y' || ans == 'Y');

    return 0;
}
```

However, 99% of the time you will use ***file I/O*** to do really thorough testing.

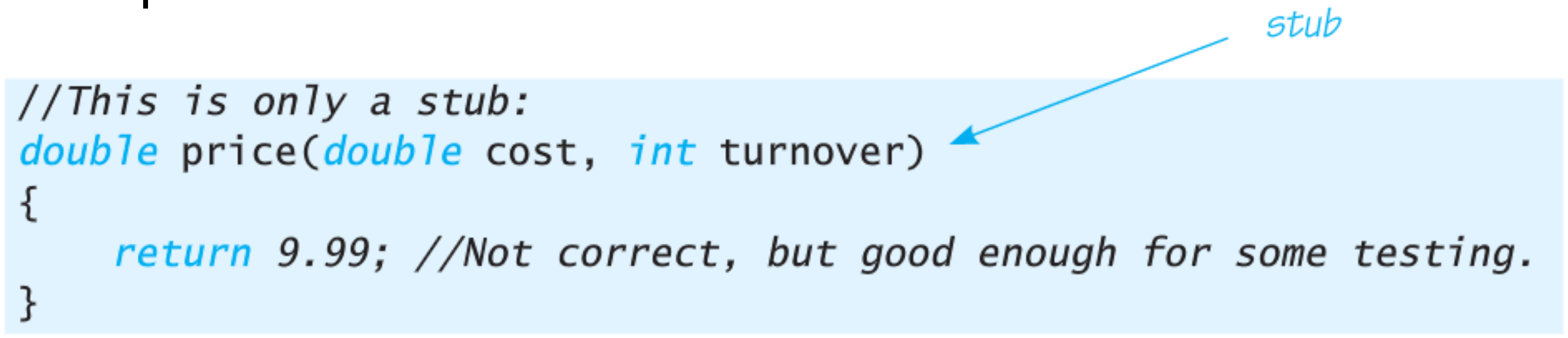
It allows you to run a batch of tests and compare the outputs offline –

You can even have the output file identify which test cases fail.

When you cannot test a function without relying on another function that has not been written/tested, we use a simplified version of the missing function.

- These simplified versions of functions are called stubs.
- They normally do not perform the correct calculation, but deliver a value that suffices for testing.
- They are also meant to be simple enough that you can be confident that they are correct.

For example:



```cpp
//This is only a stub:
double price(double cost, int turnover)
{
    return 9.99; //Not correct, but good enough for some testing.
}
```

Realistically, the price will not always be $9.99, but this is good enough to use to test other functions.

Using a top-down design approach:

- You can create stubs for all of your functions.
- Then replace the stubs one at a time with the function and test them.
- Theoretically, if you test each function as you design it and confirm that it is correct before adding a new function, you can be certain that any new bugs are part of the current function under test.
  - In reality, not necessarily so.

Alternately, you can use a driver program to create your file I/O and then combine it with the top-down approach described above.

These are all good testing and Design for Testability techniques.

But what happens when you find a bug?

How do you debug your code?

Debugging:

1. Check for Common Errors First:

- Uninitialized variables, automatic type conversion, using “=“ instead of “==“, boundary errors, etc.

2. Localize the Error:

- If you can, use test cases/cout* statements to determine where things start to go wrong. Remember in your testing you already identified at least one set of testing input that failed; you can use that as your starting place.

*Note: In some cases you will not be able to use cout/printf, so make sure that is not the only way you know how to debug.

Debugging:

3. Keep an Open Mind:

- Remember what they say about what happens when you "ASSUME". If you cannot find a bug quickly, do not assume you know where it is.

4. Do NOT randomly change sections of code hoping that will fix the error.

- This almost never works and you will never be able to localize the bug.

5. Keep track of what tests pass and fail:

- Does a different part of the code get activated when it fails? Have a log of what passed and failed so you don't keep repeating the same tests over and over again.

Debugging:

6. Assume NOTHING works (the only good assumption).

- Go through your code line by line to figure out where the problem lies.

7. Rubber Duck Debugging/ Rubber Duckie Test:

Comes from a story in the book “The Pragmatic Programmer.” In the story, the programmer explains their code line by line to the duck.

- If this doesn’t work, have someone else look at your code and explain it to them. This is the point of talking to the duck, but to learn how. The upside of talking to a person is that they are less likely to trust you (and may talk back =). The downside is you are not self-reliant).

8. Use a debugger and watch for errors in behaviour.

9. Use a memory leak checker.

More techniques to Design For Testability (DFT):

1. Use `#define DEBUG`, combined with `#ifdef DEBUG` to activate/deactivate specific regions of code only used for debugging.

- You can use it to create a “verbose” mode for your program’s operation.

2. Use the `assert` Macro. You need to `#include <cassert>` to be able to use assert macros

```
assert (boolean_expression);
```

More techniques to Design For Testability (DFT):

An example of how you can use an assert statement:

```
// Approximates the square root of n using Newton's
// Iteration.
// Precondition: n is positive, num_iterations is positive
// Postcondition: returns the square root of n
double newton_sqroot(double n, int num_iterations)
{
    double answer = 1;
    int i = 0;

    assert((n > 0) && (num_iterations> 0));
    while (i <num_iterations)
    {
        answer = 0.5 * (answer + n / answer);
        i++;
    }
    return answer;
}
```

More techniques to Design For Testability (DFT):

You can remove your assert statements from the compilation by combining the following two statements:

```
#define NDEBUG
#include <cassert>
```

If you comment out/remove the `#define NDEBUG` line, then the assert statements will be included again/turned back on”

Review Questions for Slide Set 4

- What constructor is called when you create an object that is an array of a class type?
- When do you need to use malloc/free new/delete and why?
- Are the freestore and the heap the same?
- What is a Destructor?
- What are the main differences between Destructors and Constructors?
- Are there default constructors?
- Do you have multiple constructors per class?
- What's a copy constructor? What characteristics does it exhibit?

Review Questions for Slide Set 4

Make sure you understand these characteristics about pointers::

```
int *p; //Declares a pointer to an int

int v; // Declares a variable of type int

p = &v; //Assigns p the address of v

*p = 44; // Dereferences the pointer p to assign 44 to the
         //variable v

void foo (int *value);  //The input parameter is an int pointer ->
                        //passed as a call by value; must be
                        //dereferenced to manipulate the int value

void bar (int &value);  //Call by reference, passes the address of
                         //the int to the function, but you don't
                        //need to dereference value to access the
                        //int stored in the variable.
```

Review Questions for Slide Set 4

- When do you need to include copy constructors in your class definition?
- Give an example of what might happen if you don’t have a copy constructor when you need one.
- When are copy constructors called?
- Why aren’t copy constructors called with the assignment operator?
- Assignment operators are automatically overloaded by the compiler- why would you manually need to overload one?
- Why do you need to check the special case of self assignment when overloading the assignment operator.

Review Questions for Slide Set 4

- What is a unit test?
- When should you design your tests- before or after your code? Why?
- Why use file I/O instead of user I/O to test your functions?
- What is a stub?
- What is a top-down design approach? What is a bottom-up design approach? How are they different?
- What are common bugs?
- How do you METHODICALLY debug your code?
- What are macros"
- Why use the #ifdef command?