UCLA PIC 10 B

Problem Solving using C++ Programming

- **Instructor:** Ivo Dinov, Asst. Prof. in Mathematics, Neurology, Statistics
- **Teaching Assistant:**, Suzanne Nezzar, Mathematics

University of California, Los Angeles, Summer 2001

http://www.math.ucla.edu/~dinov/10b.1.011/

Chapter 11

Pointers and Dynamic Arrays

11 Pointers and and Dynamic Arrays

11 Pointers and and Dynamic Arrays

A pointer is a construct that gives you more control of the computer's memory.

- In this chapter we discuss pointers and a new form of array called dynamic arrays.
- Dynamic arrays are arrays whose <u>size is determined while the program is running</u> rather than at writing of the program.

11.1 Pointers

- A pointer is the memory address of a variable.
- Memory is divided into adjacent locations (bytes).
- If a variable uses a number of adjacent locations, the address of the location with the smallest address is the address of the variable.
- An address that is used as to name a variable (by providing the address where the variable starts) is called a pointer variable.
- The address is said to <u>*point to*</u> a variable because it tells *where* the variable is.
- A pointer variable at 1007 can be pointed to by a pointer variable at location 2096 by supplying the address 1007, in effect, "Its over there, at 1007."
- We have used pointers in call-by-reference arguments and in array names. The C++ system handles all this automatically.

Pointer Variables (1 of 3)

- A pointer may have an address stored in it called a pointer variable.
- A pointer variable has a pointer type, and holds pointer values.
- This declares a pointer variable that can hold a pointer to double:

```
double * dPtr;
```

- This declares p1 and p2 to have type pointer to double, and v1 and v2 to have type double.

```
double *p1, *p2, v1, v2;
```

- The asterisk, *, is <u>used in two ways</u>.
- In this declaration, the asterisk, *, is a <u>pointer declarator</u>. We won't use this term much, but you should remember this for future reference.
- In the expression * p1, the asterisk is called the <u>dereferencing operator</u>, and the pointer variable p1 is said to be dereferenced. The meaning is "The value where the pointer p1 points."

Pointer Variables (2 of 3)

- **We speak of a pointer *pointing* rather than speaking of *addresses*.**
- **If pointer variable p1 contains the address of variable v1, we say that p1 points to the variable v1 or p1 is a pointer to variable v1.**
- **Given the declaration, we can make p1 point to variable v1 by:**

```
double *p1, v1;
p1 = &v1;
```

- **The & is the <u>address-of</u> operator. This statement assigns the address of v1 to the pointer variable p1.**
- **Example -- This code--**

```
v1 = 0;    p1 = &v1;    *p1 = 42;
cout << v1 << " "  << *p1 << endl;
```

Generates the output: 42 42

Pointer Variables (3 of 3)

- **We can assign the value of one pointer variable to another pointer variable of the same type:**

```
double *p1, *p2, v;
v1 = 78;     // give v1 a value
p1 = &v;     // make p1 point to v1
p2 = p1;     // assign p2 the value of p1, i.e., a pointer to v.
cout << *p2 << endl;  // output is v's value, 78.
```

- **Not all variables have to have program names:**

```
p1 = new double;  // allocates space for an double variable.
```

- **The variable created with new can only be referred to using the pointer value in p1:**

```
cin >> *p1;
```

- **Variables created using the *new* operator are called dynamic variables.**

Pointer Variable Declarations

A variable that can hold pointers to other variables of type *Type_Name* is declared the same way you declare a variable of type *Type_Name*, except that you place an asterisk at the beginning of the variable name.

Syntax:

```
Type_Name *Variable_Name1, *Variable_Name2, . . .;
```

Example:

```
double *pointer1, *pointer2;
```

Addresses and Numbers

A pointer is an address, an address is an integer, but a pointer is not an integer—That is not crazy. That is abstraction! C++ insists that you use a pointer as an address and that you not use it as a number. A pointer is not a value of type `int` or of any other numeric type. You normally cannot store a pointer in a variable of type `int`. If you try, most C++ compilers will give you an error message or a warning message. Also, you cannot perform the normal arithmetic operations on pointers. (You can perform a kind of addition and a kind of subtraction on pointers, but they are not the usual integer addition and subtraction.)

681

The * and & Operators

The * operator in front of a pointer variable produces the variable it points to. When used this way, the * operator is called the **dereferencing operator.**

The operator & in front of an ordinary variable produces the address of that variable; that is, produces a pointer that points to the variable. The & operator is simply called the **address of operator.**

For example, consider the declarations

```
double *p, v;
```

The following sets the value of p so that p points to the variable v:

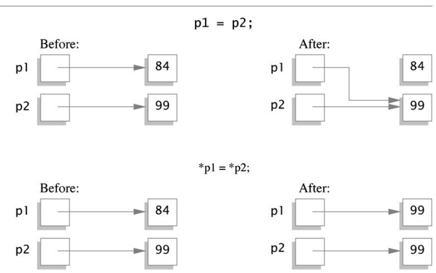
```
p = &v;
```

*p produce the variable pointed to by p, so after the above assignment, *p and v refer to the same variable. For example, the following sets the value of v to 9.99, even though the name v is never explicitly used:

```
*p = 9.99;
```


682

Display 11.1 Uses of the Assignment Operator



Display 11.2 Basic Pointer Manipulations

```
//    Program to demonstrate pointers and dynamic variables.
#include <iostream>
using namespace std;

int main( )
{  int *p1, *p2;

   p1 = new int;
   *p1 = 42;       p2 = p1;
   cout << "*p1 == " << *p1 << " *p2 == " << *p2 << endl;

   *p2 = 53;
   cout << "*p1 == " << *p1 << " *p2 == " << *p2 << endl;

   p1 = new int;    *p1 = 88;
   cout << "*p1 == " << *p1 << endl << "*p2 == " << *p2 << endl;

   return 0;
}
```

Pointer Variables Used with =

If p1 and p2 are pointer variables, then the statement

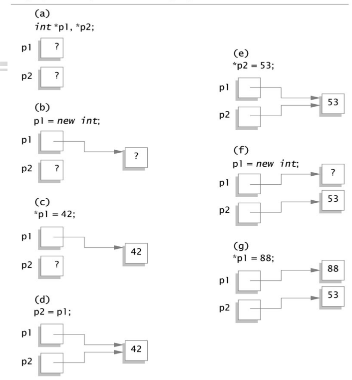
p1 = p2;

will change p1 so that <u>it points to</u> the same variable that p2 currently points to.

Unless measures are taken, any memory pointed to by p1 is lost.

685

Display 11.3 Explanation of Display 11.2



686

The *new* Operator

The *new* operator creates a new dynamic variable of a specified type and returns a pointer that points to this new variable. For example, the following creates a new dynamic variable of type MyType and leaves the pointer variable p pointing to this new variable:

```
MyType *p;
p = new MyType;
```

If the type is a class with a constructor, the default constructor is called for the newly created dynamic variable. Initializers can be specified that cause other constructors to be called:

```
int *n;
n = new int(17); // initializes n to 17
MyType *mtPtr;
mtPtr = new MyType(32.0, 17); // calls MyType(double, int);
```

With earlier C++ compilers, if there is not sufficient available memory to create the new variable, then *new* returns a special pointer named NULL. The C++ Standard provides that if there is not sufficient available memory to create the new variable, then the *new* operator, by default, terminates the program.[1]

Basic Memory Management

- **Where are variables created with the operator new?**
- **There is an area of memory called the heap, that is reserved for <u>dynamic variables</u>. The word comes from C++'s heritage in the C language. The C++ Standard and other writers use freestore instead of heap because there is an important data structure name heap. Our text follows the C heritage.**
- **Regardless of size, it is possible to consume all of heap memory with a large number of calls to new with a large data type. In this event, a program will use all heap memory then fail.**
- **A facility is provided to reclaim unused heap memory.**
- **The `delete` operator when applied to a pointer that points to memory allocated with new will release that memory, making it available for reallocation by further calls to new. It is an error to apply delete to a pointer twice and it is an error to apply delete to memory that was not allocated with new.**

688_01

NULL

NULL is a special constant pointer value that is used to give a value to a pointer variable that would not otherwise have a value. NULL can be assigned to a pointer variable of any type. The identifier NULL is defined in a number of libraries including the library with header file cstddef. With earlier compilers, the operator *new* returned a NULL pointer value whenever *new* failed in its attempt to create a dynamic variable. Current compilers "throw the exception *std::bad alloc*." The effect is to abort the program with an error message.

688_02

The *delete* Operator

The *delete* operator eliminates a dynamic variable and returns the memory that the dynamic variable occupied to the heap. The memory can then be reused to create new dynamic variables. For example, the following eliminates the dynamic variable pointed to by the pointer variable p:

```
delete p;
```

After a call to *delete*, the value of the pointer variable, like p above, is undefined. (A slightly different version of *delete*, discussed later in this chapter, is used when the dynamic variable is an array.)

Pitfall: Dangling Pointers

- If two pointers point to the same variable in the heap, and the delete operator is applied to one of them, the other still points where it did, but the memory no longer is allocated. That pointer to de-allocated memory is called a dangling pointer.
- Two remarks: The pointer that delete was applied to may or may not still be pointing where it was before deletion. The memory that was pointed to may or may not still have the same value stored in it.
- Use of a dangling pointer or a deleted pointer is very dangerous. Though illegal, few compilers detect use of such pointers.
- The worst part is that both the deleted and dangling pointers may point to the same place they did before deletion and the value stored there may not have been changed. Your program works seemingly correctly until you change some other part, then it is nearly impossible to find the error.
- Anything done to find a use of a dangling pointer is worth the effort.

Static Variables and Automatic Variables

- Variables created with new and destroyed with delete are called dynamic variables .
- Ordinary variables that we have been defining (local variables defined in a block) are called automatic variables. They are created automatically and destroyed automatically.
- Variables declared outside any function or class are called global variables. Global variables are accessible in any function after the global is defined, and in any file where the global is declared. Significant use of global variables makes code hard to understand. We do not use globals, and outside operating systems and a very few other situations, you will not need them.
- C uses the keyword static with variables defined outside any function or struct to prevent visibility from within other files. C++ has had this usage but the Standard deprecated it. (Deprecated: The next compiler version warns about deprecated usage, the next version is permitted to generate an error message). C++ uses unnamed namespaces to make names invisible outside a file.

Programming Tip:
Define Pointer Types(1 of 2)

- Writing clear code is essential. C++ provides the typedef mechanism to give a name a type value.
- With the typedef statement:

```
typedef int* IntPtr;  // make IntPtr carry  int* information
```

- The two definitions define p1 and p2 to be int pointers.

```
IntPtr p1;
int *p2;
```

- To create a typedef as an alias for a type, define an identifier with that type:

```
double *dPtr;
```

- Then place typedef in front:

```
typedef double *dPtr;
dPtr dp;
```

- dPtr carries pointer to double type.

Programming Tip:
Define Pointer Types(2 of 2)

- With these type definitions we can:
  - declare several pointer variables in one definition:
    Dptr dp1, dp2, dp3;
  - pass a pointer by reference with clarity:
    void sample(Dptr & ptr);
  - rather than writing:
    void sample(double ∗& ptr);
    // Is it ∗& or &∗ ⁇ – Typedef knows!

691

Type Definitions

You can assign a name to a type definition and then use the type name to declare variables. This is done with the keyword *typedef*. These type definitions are normally placed outside of the body of the main part of your program (and outside the body of other functions) in the same place as *struct* and class definitions. We will use type definitions to define names for pointer types, as shown in the example below.

Syntax:

typedef Known_Type_Definition New_Type_Name;

Example:

```
typedef int* IntPtr;
```

The type name IntPtr can then be used to declare pointers to dynamic variables of type *int*, as in:

```
IntPtr pointer1, pointer2;
```


11.2 Dynamic Arrays
Array Variables and Pointer Variables

- Given the following definitions,
  int a[10];
  typedef int ∗ IntPtr;
  IntPtr p;
- *a* and *p* are very close to having the same type.
- Both have the same base type.
- If p is assigned a pointer to some memory, then both may be indexed.
- p may be assigned a's value:
  p = a;
- This is how *a* and *p* are different: you CANNOT assign to *a*.
- a = p; // ILLEGAL

Display 11.4 Array and Pointer Variables

```
//Program to demonstrate that an array variable is a kind of pointer
   variable.
#include <iostream>
using namespace std;
typedef int* IntPtr;
int main( )
{   IntPtr p;
    int a[10];
    int index;
    for (index = 0; index < 10; index++)     a[index] = index;
    p = a;
    for (index = 0; index < 10; index++)  cout << p[index] << " ";
    cout << endl;
    for (index = 0; index < 10; index++)   p[index] = p[index] + 1;
    for (index = 0; index < 10; index++)   cout << a[index] << " ";
    cout << endl;
    return 0;
}
```


Creating and Using Dynamic Arrays(1 of 2)

- Size of Arrays up to this point has been defined at program writing.
- We could not set the size of an array in response to a program's need.
- Dynamic arrays using the new operator fix this problem.

```
typedef double* DoublePtr;
DoublePtr d;
int size;
cout << "Enter the size of the array" << endl;
cin >> size;
d = new double[size];
```

- To release the storage allocated for a dynamic array requires the syntax

```
delete [ ] d;
```

- to signal that the store pointed to by *d* was allocated as an array.
- Notice the position of the [ ] in this statement.

Creating and Using Dynamic Array(2 of 2)

- Do NOT attempt to release the storage allocated for a dynamic array using the syntax
  delete d;
- This is an error, but compilers do not usually detect this error.
- The Standard says the results of this is "undefined". This means the Standard allows, the compiler writer freedom, to have the compiler do *anything* convenient for the compiler writer in response to such code.
- Even if your compiler does something useful in this case, you cannot expect consistent behavior across compilers with such code.
- Always use the syntax:
  delete [ ] ptr;
- when allocation was done in a manner similar to this:
  ptr = new MyType[37];

695

How to Use a Dynamic Array

- **Define a pointer type:** Define a type for pointers to variables of the same type as the elements of the array. For example, if the dynamic array is an array of *double*, you might use the following:

  `typedef double* DoubleArrayPtr;`

- **Declare a pointer variable:** Declare a pointer variable of this defined type. The pointer variable will point to the dynamic array in memory and will serve as the name of the dynamic array.

  `DoubleArrayPtr a;`

- **Call *new*:** Create a dynamic array using the *new* operator:

  `a = new double[array_size];`

  The size of the dynamic array is given in square brackets as in the above example. The size can be given using an *int* variable or other *int* expression. In the above example, array_size can be a variable of type *int* whose value is determined while the program is running.

- **Use like an ordinary array:** The pointer variable, such as a, is used just like an ordinary array. For example, the indexed variables are written in the usual way, a[0], a[1], and so forth. The pointer variable should not have any other pointer value assigned to it, but should be used like an array variable.

- **Call *delete* [ ] :** When your program is finished with the dynamic variable, use *delete* and empty square brackets along with the pointer variable to eliminate the dynamic array and return the storage that it occupies to the heap for reuse. For example:

  `delete [] a;`

Display 11.5 A Dynamic Array (1 of 4)

```
//     Sorts a list of numbers entered at the keyboard.
#include <iostream>
#include <cstdlib>
#include <cstddef>
using namespace std;

typedef int* IntArrayPtr;

void fill_array(int a[ ], int size);
//Precondition: size is the size of the array a.
//Postcondition: a[0] through a[size-1] have been
//filled with values read from the keyboard.

void sort(int a[ ], int size);
//  Precondition: size is the size of the array a.
//      The array elements a[0] through a[size-1] have values.
//  Postcondition: The values of a[0] through a[size-1] have been
//     rearranged  so that a[0] <= a[1] <= ... <= a[size-1].

void swap_values(int& v1, int& v2);
int index_of_smallest(const int a[], int start_index, int number_used);
```


Display 11.5 A Dynamic Array (2 of 4)

```
int main( )
{   cout << "This program sorts numbers from lowest to highest.\n";

    int array_size;
    cout << "How many numbers will be sorted? ";
    cin >> array_size;

    IntArrayPtr a;
    a = new int[array_size];

    fill_array(a, array_size);
    sort(a, array_size);
    cout << "In sorted order the numbers are:\n";

    for (int index = 0; index < array_size; index++)
        cout << a[index] << " ";
    cout << endl;

    delete [ ] a;                  // ← Do NOT forget to release system resources

    return 0;
}
```

Display 11.5 A Dynamic Array (3 of 4)

```
//Uses the library iostream:
void fill_array(int a[ ], int size)
{
   cout << "Enter " << size << " integers.\n";
   for (int index = 0; index < size; index++)
      cin >> a[index];
}

void sort(int a[ ], int size)
{  int index_of_next_smallest;
   for (int index = 0; index < size - 1; index++)
   {   //      Place the correct value in a[index]:
      index_of_next_smallest =
               index_of_smallest(a, index, size);
      swap_values(a[index], a[index_of_next_smallest]);
      // a[0] <= a[1] <=...<= a[index] are the smallest of the original array
      // elements. The rest of the elements are in the remaining positions.
   }
}
```


Display 11.5 A Dynamic Array (4 of 4)

```
void swap_values(int& v1, int& v2)
{  int temp;
   temp = v1;
   v1 = v2;
   v2 = temp;
}

int index_of_smallest(const int a[], int start_index, int number_used)
{  int min = a[start_index],
   index_of_min = start_index;
   for (int index = start_index + 1; index < number_used; index++)
    { if (a[index _of_min] < min)
      {
        min = a[index];
        index_of_min = index;
        //min is the smallest of a[start_index] through a[index]
      }
    }

   return index_of_min;
}
```


Pointer Arithmetic (1 of 2)

- You <u>can do arithmetic with pointers and integers</u>. The behavior is easy to understand.
- Consider the code:
  typedef double * DoublePtr;
  DoublePtr dPtr;
  dPtr = new double[10];
- At this point in the code, dPtr contains the address of indexed variable dPtr[0].
- The expression <u>dPtr + 1 contains the address of dPtr[1]</u>
- The expression <u>dPtr + 2 contains the address of dPtr[2]</u>
- . . .
- What is happening?

Pointer Arithmetic (2 of 2)

- What is happening is:
- The compiler knows the size of a double.
- When the program adds 1 to dPtr, the compiler generates code to add the size of a ONE double to dPtr.
- This generates the address of dPtr[1].
- When the program adds 2 to dPtr, the compiler generates code to add 2 times the size of a double to dPtr.
- This is the address of dPtr[2].
- In these loops that step through a dynamic array generate the same output:

```
for (int i = 0; i < size; i++)
    cout <<*(dPtr + ii) << " " ; // + overload
for (int i = 0; i < size; i++)
    cout << dPtr[i] << " " ;
```


11.3 Classes and Dynamic Arrays

- A dynamic array, like an ordinary array, can have a class or struct type as a base type.
- A class or a struct can have a dynamic array as a member.
- The basic techniques are exactly as you expect.
- However, there are some details when using classes and dynamic arrays that, if neglected, can cause a disaster.

Programming Example:
A String Variable Class (1 of 3)

- We talked about the Standard string type in Chapter 10, so we don't need to write our own string class.
- Nevertheless it is an excellent exercise to design and code a string class. See Display 11.6 for the interface.
- There are four StringVar constructors and a destructor.
  - An int parameter constructor that creates an empty StringVar of size equal to the constructor's argument.
  - A default constructor that creates an empty StringVar with store allocated of size 100 characters.
  - A constructor that creates StringVar object with the characters from a cstring argument.
  - A copy constructor so we can pass a StringVar object to a function as value parameters, return our string from a function, and initialize one StringVar object from another StringVar object.

Programming Example:
A String Variable Class (2 of 3)

- There is a destructor, ~StringVar( ) to release dynamically allocated memory to the heap manager.
- Details of the destructor are presented later.
- See Display 11.7 for a simple demonstration program.
- The constructors allocate a dynamic array of size depending on the constructor. The StringVar object created is empty except for the cstring constructor.
- The constructor with the int parameter allocates a dynamic array of size equal to the argument, and sets max_length to this value.
- The default constructor allocates a dynamic array of size 100, and sets max_length to 100.
- The constructor with a cstring parameter allocates a dynamic array of size equal to the argument size, and sets max_length to this value.

Programming Example:
A String Variable Class (3 of 3)

- The StringVar class is implemented using a dynamic array. The implementation is in Display 11.8.
- At definition of a StringVar object, a constructor is called that defines a dynamic array of chars using the `new` operator and initializes the object.
- The array uses the null character, '\0', to indicate "past the last" character as is done in a cstring.
- Note that StringVar indicates end of string differently from String from Display 10.11. There a separate int value is used to record length. There are trade-offs, as in everything in Computer Science and Information Systems.

Display 11.6 Interface file for StringVar class (1 of 3)

```
//  FILE strvar.h
//  This is the INTERFACE for  class StringVar whose
// values are strings.  Note that you use (max_size), not
//  [max_size]  StringVar  Your_object_Name(max_size);
//  max_size is the longest string length allowed.
//  max_size can be a variable
#ifndef STRVAR_H
#define STRVAR_H
#include <iostream>
using namespace std;
namespace savitchstrvar
{   //   class StringVar
```


Display 11.6 Interface file for StringVar class (2 of 3)

```
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 heap.
```


Display 11.6 Interface file for StringVar class (3 of 3)

```
    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 already 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 the dynamic array that holds the string value.
    int max_length; //declared max length of any string value.
  };
}          // savitchstrvar
#endif     // STRVAR_H
```


Display 11.7 Program using StringVar class

```
#include <iostream>
#include "strvar.h"
using namespace std;
using namespace savitchstrvar;

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

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

//     This is only a demonstration function:
void conversation(int max_name_size)
{  StringVar your_name(max_name_size), our_name("PIC10B");

   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;
}
```

Display 11.8 Implementation of StringVar (1 of 3)

```
// FILE: strvar.cpp          IMPLEMENTATION of the class StringVar.
#include <iostream>
#include <cstdlib>
#include <cstddef>
#include <cstring>
#include "strvar.h"
namespace savitchstrvar
{
   //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';
   }
```


Display 11.8 Implementation of StringVar (2 of 3)

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


Display 11.8 Implementation of StringVar (3 of 3)

```
    //    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;
    }

 }    //    savitchstrvar
```


Destructors (1 of 3)

- **A dynamic variable is ONLY accessible through a pointer variable that tells where it is. Their memory is not released at the end of the block where the local (automatic) variable was created. Memory allocated for dynamic variables <u>must be released by the programmer.</u>**
- **This is true even if the memory is allocated for a local pointer to point to, and the pointer goes away. The memory remains allocated, and deprives the program and the whole computer system of that memory until the program that allocated it stops.**
- **For a badly behaved programs, this can cause the program or maybe the operating system to crash.**

Destructors (2 of 3)

- **If the dynamic variable is embedded in the implementation, a user cannot be expected to know, and cannot be expected to do the memory management, even in the unlikely event that facilities for such are provided.**
- **The good news is C++ has destructors that are implicitly called when a class object passes out of scope.**
- **If in a function, you have a local variable that is an object with a destructor, when the function ends, the destructor will be called automatically.**
- **If defined correctly, the destructor will do what ever clean-up the programmer intends, part of which is deleting dynamic memory allocated in by the object's constructors.**

Destructors (3 of 3)

- **A destructor's name is required to be the name of the class, except the class name is prefixed by the tilde character, ~.**
- **The member ~StringVar of the class StringVar is the destructor for this class.**
- **Examine the implementation of ~StringVar, and notice that it calls delete to release the dynamic memory to the heap manager**

Destructor

A **destructor** is a member function of a class that is called automatically when an object of the class goes out of scope. Among other things, this means that if an object of the class type is a local variable for a function, then the destructor is automatically called as the last action before the function call ends. Destructors are used to eliminate any dynamic variables that have been created by the object so that the memory occupied by these dynamic variables is returned to the heap. Destructors may perform other clean-up tasks as well. The name of a destructor must consist of the tilde symbol ~ followed by the name of the class.

Pitfall

Pointers as Call-by-Value Parameters (1 of 2)

- If a call-by-value parameter is a pointer, the behavior can be subtle and troublesome.
- If a pointer call-by-value parameter is dereferenced inside a function, the dereferenced pointer expression can be used to fetch the value of the variable the pointer points to, or the expression can be used to assign a value to the variable the pointer points to.
- This is exactly the scenario in function void (IntPointer sneaky) in Display 11.9.
- There temp is a local variable, and no *changes to temp* go outside the function. This does not extend to an expression that is a *dereferenced pointer* parameters.
- Dereferencing the pointer, temp, that is a copy of the argument that points to a variable in main will make that variable accessible inside the function.

Pitfall

Pointers as Call-by-Value Parameters (2 of 2)

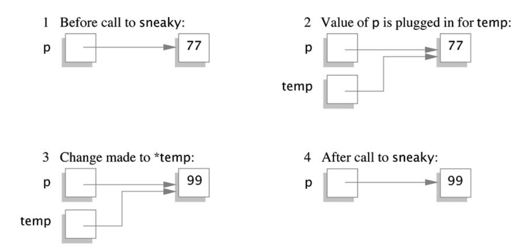
- If the parameter is struct or class object with a member variable of a pointer type, changes can occur with a call-by-value parameter.
- Inadvertent and surprising changes can be controlled by writing copy constructor for classes.

Display 11.9 A Call-by-Value Pointer Parameter

```
//    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;
}
```


710

Display 11.10 The Function Call sneaky(p);



Copy Constructors (1 of 8)

- A copy constructor is a constructor that has one parameter that is a reference to an object of the same type as the class.
- In order to be able to copy const objects, the copy constructor usually has a const reference parameter.
- The reference parameter (&) is to break the implied infinite recursion that would otherwise occur with the copy constructor.
- Historical Note: With an early C++ compiler from a well known company, if the & was omitted in the copy constructor, the result was an out of memory system crash during compilation.
- A copy constructor's purpose is just as the name implies: to construct an object that is a copy of the argument object.

Copy Constructors (2 of 8)

- Example:

```
StringVar line(20), motto("Constructors help!");
cout << "Enter a string of length 20 or less:\n";
line.input_line(cin);
StringVar temp(line);  // copy constructor creates temp as duplicate
                        // of object line.
```

- The constructor used is selected by the compiler based on the argument.
- In the first line, the argument 20 is an exact match for the int parameter constructor.
- In the second constructor, the "Constructors… " argument is an exact match for the const char[ ] parameter.
- In the last line, the argument is a StringVar object, which calls the copy constructor.

Copy Constructors (3 of 8)

- We have pointed out in these slides that a copy constructor is called in several situations.

1. Any time C++ needs to make a copy of an object, the copy constructor is called automatically. These situations are:
2. When a class object is being defined and initialized by another object of the same type,
3. When a class object is the return value of a function,
4. when a class object is plugged in for a call-by-value parameter. The copy constructor defines what "plugged in for" means.

Copy Constructors (4 of 8)

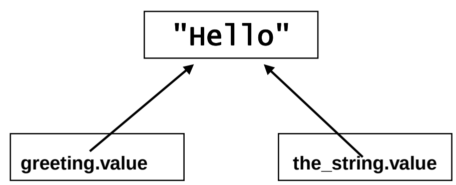
- If there is no copy constructor, the members are copied according to the default for the member:
  - Built-in types are just copied, which is fine.
  - Pointers are just copied too, which isn't "fine". You have two pointers to the same memory. An example follows.
  - Copying fails members declared to be arrays.
  - Members that are class objects are also copied, using the copy constructor for that class.
- This is called Member-Wise copy. A student coined the phrase, "Member UN-wise copy". Let's see why.

Copy Constructors (5 of 8)

- Suppose that in the StringVar class, there is no copy constructor, but there IS a destructor. Consider this code:

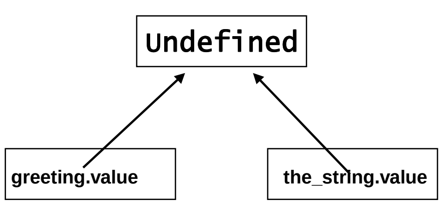
```
void show (StringVar the_str)
{
   cout <<"The String is: " << the_string << endl;
}

//Suppose in another function we have this code:
 StringVar greeting("Hello");
 show(greeting);
 cout << "after the call: " << greeting << endl;
```


Copy Constructors (6 of 8)



This is the situation before the function ends.
Since we used a call-by-value both greeting and the_string are pointers to the same memory location.

Copy Constructors (7 of 8)



This is the situation after the function ends. The destructor has been called, invoking
delete[ ] the_string;
which makes the memory pointed to by greeting.value and the_string.value have an undefined value.

Copy Constructors (8 of 8)

- Object passed to a function by value was destroyed.
- The destructor was invoked on a member-(un)wise COPY of the object, which in turn destroyed the data common to the local copy object and the argument object .
- In contrast with many programming languages (Java in particular) where the semantics of initialization and assignment are identical, C++ make careful distinction between initialization (done by the copy constructor) and assignment (done by operator assignment overloading).
  That's why with cstring:: char A[10]="ABC"; works, but A="ABC"; doesn't!
- Initialization is done by the copy constructor which creates a new object that is an identical copy of the argument.
- The assignment operator modifies an already existing object into a copy that is identical in all respects except location to the right-hand side of the assignment .

714_01

Copy Constructor

A **copy constructor** is a constructor that has one call-by-reference parameter that is of the same type as the class. The one parameter must be a call-by-reference parameter; and normally the parameter is also a constant parameter, i.e., preceded by the *const* parameter modifier. The copy constructor for a class is called automatically whenever a function returns a value of the class type. The copy constructor is also called automatically whenever an argument is "plugged in" for a call-by-value parameter of the class type. A copy constructor can also be used in the same ways as other constructors.

Any class that uses pointers and the *new* operator should have a copy constructor.

714_02

The BIG Three

The **copy constructor,** the = **operator,** and the **destructor** are called the **big three** because experts say if you need any of them you need all three. If any of these is missing, the compiler will create it, but it may not behave as you want. So it pays to define them yourself. The copy constructor and overloaded = operator that the compiler generates for you will work fine if all member variables are of predefined types such as *int* and *double*, but it may misbehave on classes that have class member variables. For any class that uses pointers and the new operator, it is safest to define your own copy constructor, overloaded =, and a destructor.

Caveat: Good design says you need all three or none. Clearly, you can omit any one of these that you can guarantee you are not going to use.

Overloading the Assignment Operator(1 of 4)

- **If String1 and String2 are defined as follows:**

  StringVar sring1(10), sring2(20);

  Suppose further that string2 has been given a value, this assignment is defined, but the default definition is NOT defined in StringVar:

  string1 = string2;
- **Like the copy constructor, the default operator assignment copies members. The effect is as if we had access the private members and these assignments were carried out:**

  string1.value = string2.value;

  string1.max_length = string2.max_length;

  The pointer members of string1 and string2 share the data that belonged only to string2 before the assignment. This is member-wise copy.

Overloading the Assignment Operator(2 of 4)

- **How do we fix this problem? Answer: We overload the = operator.**
- **<u>Operator = is one of four operators that must be overloaded as regular members of a class; they cannot be overloaded as a friend.</u>**
- **class StringVar should be changed as follows:**

```
class StringVar
{
  public:
    void operator=(const StringVar & rhs);
    // the remainder is the same as Display 11.6
};
```

- **Assignment is carried out just as we indicated earlier:**

  string1 = string2;
- **As in all operator overloading, this infix is converted to a call to the operator= overloading function with the left hand member of the assignment is the calling object, the right hand side is the argument.**

Overloading the Assignment Operator(3 of 4)

- **When we implement operator =, we should check for unobvious errors such as destroying the left hand side too soon. This would cause a bug when the rhs and rhs are the same object, as in**

  string1 = string1;
- **We need to decide whether there is enough room in the left hand side string to store the right hand side string. If not, they aren't the same string, destroy the left hand side, allocate enough space then copy.**
- **If there is enough space we don't need to destroy lhs object, so we proceed to copy the rhs object to the char array of the lhs.**
- **You should note that our implementation returns void. This means only that we <u>cannot write a chain of assignments</u>, as in**

  string1 = string2 = string3;

Implementing this involves changing the return type to StringVar and returning the right hand side of the assignment. We leave this as an assignment for the interested student.

Overloading the Assignment Operator(4 of 4)

```
// Final version of implementation
void StringVar::operator=(const StringVar & rhs)
{
  int new_length = strlen(rhs.value);
  if (new_length > max_length)  //not enough room in lhs
  {
    delete [ ] value;          //deallocate lhs space
    max_length = new_length;
    value = new char[max_length + 1];//allocate space
  }
  for(int i = 0; i < new_length; i++)//have space now,
     value[i] = rhs.value[i];        //copy data.
  value[new_length] = '\0';
}
```

CHAPTER SUMMARY

719-20 ■ A **pointer** is a memory address, so a pointer provides a way to indirectly name a variable by naming the address of the variable in the computer's memory.

- **Dynamic variables** are variables that are created (and destroyed) while a program is running.
- Memory for dynamic variables is in a special portion of the computer's memory called the **heap.** When a program is finished with a dynamic variable, the memory used by the dynamic variable can be returned to the heap for reuse; this is done with a `delete` statement.
- A **dynamic array** is an array whose size is determined when the program is running. A dynamic array is implemented as a dynamic variable of an array type.
- A **destructor** is a special kind of member function for a class. A destructor is called automatically when an object of the class passes out of scope. The main reason for destructors is to return memory to the heap so the memory can be reused.
- A **copy constructor** is a constructor that has a single argument that is of the same type as the class. If you define a copy constructor it will be called automatically whenever a function returns a value of the class type and whenever an argument is "plugged in" for a call-by-value parameter of the class type. Any class that uses pointers and the operator `new` should have a copy constructor.