Unit 8

Object-Oriented Programming in C++

Data Encapsulation

CONTENTS

Objectives

At the end of this unit we will be able to:

- list some benefits of data encapsulation
- provide complete data encapsulation for a class with
  - — an assignment operator
  - — a constructor
  - — a destructor
  - — a "copy" constructor

Data Encapsulation

There are many advantages to using a language that provides data encapsulation. Since the representation of an object type can only be manipulated by a limited set of operations (those listed in the object's class), only that limited set of operations depends on the representation. If we change the representation, we only need to re-write the class's defining operations.

Data encapsulation can also allow the author of a class to enforce rules about consistency within the class's private data. The author of a class for doubly linked lists might enforce the rule that the `previous` pointer of the next element in the list must point back to this object. The author of a class representing sets might enforce the rule that no element can appear twice in the a set. A rule that will always be true for any "legal" object is often called a *representation invariant.*

If a class's defining operations generate only objects that obey the representation invariant, there is no way for a user of the class to create an illegal object. Therefore, all the class's operations can be coded with the assumption that the objects they work with must have started in a legal state. Thus, the enforcement of a representation invariant is a powerful tool for managing subtle dependencies between the data members of a class, without fear that users will create objects that are not consistent with the author's rules.

The representation invariant for our class `String` is described by the comment in the private section. Note that many of our member functions are written with the assumption that the `text` array will be null terminated.

```
class String {
    //...

private:
    // a String is a sequence of up to
    // max_string_length non-null characters
    // followed by a null character

    char text[max_string_length+1];
};
```

Data Encapsulation

Benefits of data encapsulation

- limits implementation dependence
- facilitates change of implementation
- ensures consistency of data
- allows control of subtle dependencies between data

but only if functions listed in class are the only ones that depend on implementation.

Built-in Operations

A language supports data encapsulation if only the operations used in the definition of a class can access the implementation of the class. In C, all objects (including structure type objects) can be created (e.g., with a variable declaration), destroyed (e.g., at the end of the function in which they were declared), assigned, and passed to functions or returned.

C++ resolves the conflict between data encapsulation and the legal C operations with the following rule: The C definitions of assignment (by copying the object), creation (allocation of un-initialized memory), destruction (freeing of memory), and initialization (by copying the object) are built into the C++ language as defaults, but a class may take control of them.

If the built-in mechanisms for assignment, creation, destruction, and initialization do not allow the user to create objects that violate the representation invariant, then the author of a class need not worry about taking control of these operations. If, however, one of these built-in mechanisms allows the creation of an illegal object, the author of the class must override the built-in mechanism by providing code that is appropriate for the class.

Built-in Operations

C++ allows several operations not listed in the class:

- assignment
- creation (variable declaration)
- destruction (at end of functions)
- initialization w/ same type

```
#include "String.h"

main(int, char *[])
{
    // Object creation:
    String s, t;
      // use of operator=(const char *):
      s = "Zaphod Beeblebrox";

    // Initialization:
    String u = s;

    // Assignment of same type:
    t = s;

    // Object Destruction (automatic upon exit from block)
    return 0;
}
```

Controlling Assignment

You can control the assignment of objects of a class by creating a special `operator=` for that class. If class `String` declares `operator=(String)`, then that operator will be used to handle assignment of one string to another. If class String does not declare this operator, then C++ will copy all the data members from the `String` on the right into the `String` on the left (as is done with structure assignment in C).

```
String s1, s2;

s1 = "hello world";  // String::operator=(char *)
s2 = s1;         // String::operator=(String),
                 // or default mechanism for compatibility
```

Even though the default mechanism for assignment will work for our class `String`, we'll override that default in our class, to show how it is done. Later, we will see an example class for which the default mechanism for assignment creates an illegal object.

If you wish to make assignment illegal for objects of your class, declare the `operator=` as a private member function of the class. Users will not be able to assign objects, since that would require access to the private member function.

Controlling Assignment

```
const int max_string_length = 128;

class String {

public:
  String &operator=(const String &);
     String &operator=(const char *);
     int  length() const;
     int  read();
     void print() const;

     const char &operator [](int) const;
     char & operator [](int);
     String substring(int start, int len) const;

     friend String operator+(const String &, const String &);
     friend String operator+(const String &, const char *);
     friend String operator+(const char *, const String &);

private:
     // a String is a sequence of up to
     // max_string_length non-null characters
     // followed by a null character

     char text[max_string_length+1];
};
```

String Assignment

Note that our `operator=` copies only the bytes before the null, but the default mechanism copies all `max_string_length + 1` bytes.

String Assignment

```
#include "String.h"
#include <string.h>

// use: s = "hello world"

String &String::operator=(const char *rhs)
{
     strncpy(text, rhs, max_string_length);
     text[max_string_length] = '\0';

     return *this;
}

// use: s = t

String &String::operator=(const String &rhs)
{
   strcpy(text, rhs.text);

     return *this;
}
```

Constructor and Destructor Functions

To ensure that all objects always satisfy the representation invariant stated in their class, we must ensure that objects are created in a legal state, and that no legal object can ever be corrupted.

To ensure that objects are created in a legal state, the author of a class can write a function that will control the creation of objects. C++ provides a special kind of member function, known as a *constructor* function, for this purpose. If a class has one or more constructor functions, C++ will automatically call one of them whenever an object of that class type is created (unless C++ is using the built-in rule for initialization with an object of the same type). The constructor function is used *after* the memory for the data members has already been allocated. The constructor does *not* have to allocate this storage.

To ensure that no legal object is ever corrupted, we need only ensure that none of the class's defining operations can change a legal object into an illegal one. Objects may enter an illegal state temporarily, during the execution of the defining operations, as long as (1) they have been restored to a legal state by the end of the function, and (2) no other functions that depend on the representation invariant will be called while the object is in an illegal state.

If only legal objects can be created, and there is no way to make a legal object into an illegal one, then all objects will always satisfy the representation invariant stated in their class[1]. When an object is destroyed, the mechanism that destroys it must be able to handle any legal object. If the representation invariant states that the object has a pointer to some additional storage, that storage may need to be freed, so the rule "free the storage for the data members" (used by C for structures) is not always correct for class-type objects. To ensure that any legal object will be destroyed properly, the author of a class can write a function that will control the destruction of objects. C++ provides a special kind of member function, known as a *destructor* function, for this purpose. If a class has a destructor, C++ will automatically call it whenever an object of that class type is destroyed. The destructor function is used *before* the memory for the data members has already been freed (the destructor does *not* have to free this storage).

If a class has no constructors, you may still create objects of that class. C++ will simply allocate enough memory for the data members in the class (as C would do for a **struct** type variable). This memory may not be initialized, so there is no guarantee that objects created in this way will obey the class's representation invariant. Note that this is not a problem for built-in types (e.g., `int`), since any piece of memory of the appropriate size is a legal object.

If a class has no destructor, you can still destroy objects of that type. C++ will simply free the memory for the data members (as C would do for a **struct** type variable).

1. In a language that allows unrestricted use of pointers, we can never be absolutely sure of the state of objects in memory, since a programmer could corrupt any value by making a mistake with pointers. So when we say that objects will always be legal, we mean they will be legal unless there is a pointer mistake in the program.

Constructor and Destructor Functions

Constructor functions

- member functions with the same name as the class
- automatically called when an object is created, just after memory allocation
- allow the class to control object creation

Destructor functions

- member functions named ~*class-name*
- automatically called when an object is destroyed, just before memory is freed
- allow the class to control object destruction

```
func() {
     String s;
     // constructor called automatically

     // work with s...

}    // destructor called automatically
```

Declaring Constructors & Destructors

Constructor functions for class `String` are member functions named `String`. We will see that a class can have many constructors, to handle the creation of objects with different types of initializers. The destructor function is named `~String`. Since these functions are called automatically, we can not declare a return type (not even **void**).

Declaring Constructors & Destructors

```
const int max_string_length = 128;

class String {

public:
   String();
   ~String();

     String &operator=(const String &);
     String &operator=(const char *);
     int  length() const;
     int  read();
     void print() const;

     const char &operator [](int) const;
     char & operator [](int);
     String substring(int start, int len) const;

     friend String operator+(const String &, const String &);
     friend String operator+(const String &, const char *);
     friend String operator+(const char *, const String &);

private:
     // a String is a sequence of up to
     // max_string_length non-null characters
     // followed by a null character

     char text[max_string_length+1];
};
```

Defining Constructors & Destructors

When an object is created, the constructor will be called just after the storage has been allocated. The newly allocated storage will be used as the invoking object for the constructor, so when the constructor puts a null byte at the beginning of the invoking object's `text`, it is working with the newly created `String`. This will ensure that the representation invariant for class `String` holds for the newly created object.

When an object is destroyed, the destructor will be called just before the storage is freed. The object being destroyed will be the destructor's invoking object. The class `String` doesn't need to do anything when a `String` is destroyed, so we have provided an empty destructor. We could have left out the destructor, as it is legal to have a constructor with no destructor, or a destructor with no constructor. Later, we will see an example of a class that requires a destructor.

Defining Constructors & Destructors

```
#include "String.h"

//
// constructor function
// called automatically when a string is created.
//
// ensure the string is null terminated
//

String::String()
{
    text[0] = '\0';
}

//
// destructor function
// called automatically when a string is destroyed.
//

String::~String()
{
    // nothing needed here yet
}
```

Calls to Constructors & Destructors

Constructors and destructors are used to create objects of all storage classes.

External and nonlocal static objects, such as `extrn` and `statc` on the facing page, will be created before they are used. Typically, they are all created before the execution of the first statement in `main`. Thus, constructors may be run before the beginning of `main`. The destructors for external and nonlocal static variables will be called upon return from `main`, or when `exit` is called.

Automatic and local static objects, such as `autom` and `loc_statc`, are created when the flow of control reaches the variable definition. Therefore, the `String` constructor will be called for `autom` and `loc_statc` after the first call to `printf` in `example`. Note that, in C++, declarations may be interspersed with statements.

Automatic objects are destroyed upon exit from the block in which they were created, so the destructor for `autom` will be called just after the second call to `printf` in `example`. When the `example` function is called the second time, `autom` will be re-created, and the `String` constructor will be run again.

Static objects last until the end of the program, so the destructor for `loc_statc` will not be called until `exit` is called or the program returns from `main`. When the `example` function is called the second time, the constructor will not be called for `loc_statc`, because `loc_statc` will already exist. If the `example` function had not been called by our `main` program, and `loc_statc` had not been created, then the destructor would not be called for it.

The constructor and destructor will be used for each element in an array, if the elements are of a class with a constructor and destructor. The constructor is used on each element, in order of increasing addresses, when the array is created. The destructor is used when the array is destroyed, and is applied to the elements in order of decreasing addresses.

Objects can also be created on the free store with the **new** operator, and destroyed with the **delete** operator (these operators will be discussed in a later unit -- they are used for management of free store, as the functions `malloc` and `free` are used in C). When such objects are created and destroyed, their constructors and destructors are used.

Calls to Constructors & Destructors

```
#include "String.h"
#include <stdio.h>

String extrn;
static String statc;

void example()
{
    printf("entering example function.\n");

    String autom;
    static String loc_statc;

    String array[30];

    printf("leaving example function.\n");
}

main(int, char *[])
{
    printf("start of program.\n");
    example();
    example();
    printf("end of program.\n");
}
```

Initialization

In C++, the word "initialization" is used to describe code that provides a value for an object during the creation of that object (e.g., in a declaration). The word "assignment" is used for code that stores a value in an object that already exists.

```
int i = 7;  // initialization
int k;

k = 7500000; // assignment
i = 0;       // assignment
```

C++ uses constructor functions for initialization, and assignment operators for assignment.

There are two ways to initialize an object of a class type: with the '=' symbol (as in C), or with a list of initial values in parenthesis. The second syntax is needed when an object must be initialized with more than one value, and it can only be used with class types:

```
#include "complex.h"

        complex c = 7.4;       // initialization 7.4 + 0i
        complex d(8.5);        // initialization 8.5 + 0i
        complex e(9.1, 10.0); // initialization 9.1 + 10.0i
        complex f;
```

When a class has constructor functions, you must provide a constructor function to handle the case of a declaration that does not provide an initial value[2], if you want that case to remain legal. In other words, if class `String` has a constructor `String(const char *)` but no constructor `String()`, then `String s =" hello ";` is legal, but `String s;` is not.

If a class does not provide a constructor to create an object from an initializing value of the same type[3], C++ will copy the data members from the initial value into the object being created (as C would have done with a **struct** type variable). For example, if there were no constructor `String(const String &)`, C++ would still allow `String t = s;`, for compatibility with C (but if the constructor exists, C++ will use it). If you wish to make this form of initialization illegal for objects of your class, declare the copy constructor as a private member function of the class. Users will not be able to initialize objects with values of the class type, since that would require access to the private member function.

2. A constructor that has no parameters (and therefore is used to create objects when no initial value is given) is sometimes called a *default constructor*.
3. A constructor to initialize an object with a value of the same type is sometimes called a *copy constructor*

Initialization

```
#include "String.h"
#include <stdio.h>

main(int, char *[])
{
    String output;
         // String::String()

    String firstname = "Irving";
    String middleinit("J.");
         // String::String(const char *)

    String lastname = firstname;
    String name(firstname + " " + middleinit + " " + lastname);
       // String::String(const String &)
       // (or copies data members)

    output = "name is: " + name;
         // operator=(const String &)

    output.print();

    return 0;
}
```

Additional Constructor Functions

The constructor functions shown on the facing page will allow the initialization of `Strings` with either `String` values or `char *` values.

An example of multiple-value initialization is shown below:

```
class complex {
public:
    complex();
    complex(float real_part);
    complex(float real_part, float imaginary_part);

private:
    float real, imag;
// or: float angle, distance;
};

#include "complex.h"

     complex c = 7.4;      // initialization 7.4 + 0i
     complex d(8.5);       // initialization 8.5 + 0i
     complex e(9.1, 10.0); // initialization 9.1 + 10.0i
     complex f;
```

Additional Constructor Functions

```
const int max_string_length = 128;

class String {

public:
     String();
   String(const char *);
   String(const String &);
     ~String();

     String &operator=(const String &);
     String &operator=(const char *);
     int  length() const;
     int  read();
     void print() const;

     const char &operator [](int) const;
     char & operator [](int);
     String substring(int start, int len) const;

     friend String operator+(const String &, const String &);
     friend String operator+(const String &, const char *);
     friend String operator+(const char *, const String &);

private:
     // a String is a sequence of up to
     // max_string_length non-null characters
     // followed by a null character

     char text[max_string_length+1];
};
```

Defining Additional Constructor Functions

The new constructors are shown on the facing page. Note that both ensure that the `text` member of the `String` being created will be null terminated.

Defining Additional Constructor Functions

```
#include "String.h"
#include <string.h>

//
// constructor function
// called automatically when a string is created
// and initialized with a char* value
//

String::String(const char *init)
{
    strncpy(text, init, max_string_length);
    text[max_string_length] = '\0';
}

//
// constructor function
// called automatically when a string is created
// and initialized with a String value
//

String::String(const String &init)
{
    strcpy(text, init.text);
}
```

Summary

Summary

A class can control all the operations that access its private data:

- All member functions
- All friend functions
- Operations that were legal in C
  - Assignment
    ```
    String::operator=(const String &)
    ```
  - Creation
    ```
    String::String() // "default" constructor
    ```
  - Destruction
    ```
    String::~String()  // destructor
    ```
  - Initialization
    ```
    String::String(const String &)
         // "copy" constructor
    ```

replace with blank page

Class-type Members

Objectives

Objectives

At the end of this appendix we will be able to:

- create a class with a class-type data member
- create for that class:
  - — an assignment operator
  - — a constructor
  - — a destructor
  - — a "copy" constructor
- understand what will happen without the above functions

Class-type Data Members

A class may have class-type data members. In this case, the class of the data member must come first in the source file.

Class-type Data Members

```
#include "String.h"

class Employee {
public:
    void set_name(const String &);
    String get_name() const;

    void set_salary(float);
    float get_salary() const;
    //...

private:
    String name;
    float salary;
};
```

Built-in Operations

The creation of an `Employee` includes the creation of that `Employee's name`, which is a `String`. Since class `String` has constructor functions, a `String` constructor must be called whenever a `String` is created, even if that `String` is part of another object. When an `Employee` object is created, memory is allocated, and the constructor `String::String()` is called. Later, we will see how class `Employee` can select one of the other `String` constructors.

If a class does not provide an assignment operator, C++ will assign all the data members from the object on the right to the corresponding members of the object on the left. If any of the data members has an `operator=` that handles assignment of that type of object (e.g., `String::operator=(const String &)`), then that member's assignment operator will be used to assign that member. Therefore, when the `Employee e1` is assigned to `e2`, `String::operator=(const String &)` will be used to assign `e1.name` to `e2.name`.

If a class does not provide a copy constructor, C++ will copy all the data members from the initializer to the corresponding members of the object being created. If any of the data members has a copy constructor, then that copy constructor will be used to initialize that member. Therefore, when the `Employee e3` is initialized with the value `e2`, `String::String(const String &)` will be used to initialize `e3.name` with the value `e2.name`.

The destruction of an `Employee` includes the destruction of that `Employee's name`, which is a `String`. Class `String's` destructor must be called whenever a `String` is destroyed, even if that `String` is part of another object. When an `Employee` object is destroyed, the `String` destructor will be used to destroy that `Employee's name` before the memory for the `Employee` has been released.

Built-in Operations

```
#include "Employee1.h"

main(int, char *[])
{
     String jones = "Jones";

     Employee e1, e2;   // String::String()
                       // used on e1.name and e2.name

     e1.set_name(jones);
     e1.set_salary(30000);

     e2 = e1;   // String::operator=(const String &)
              // used to assign e1.name to e2.name

     Employee e3 = e2;   // String::String(const String &)
                       // used to copy e2.name to e3.name

     return 0;   // String::~String()
              // used for e1.name, e2.name, and e3.name
}
```

Overriding Built-in Operations

Class `Employee` can provide its own assignment operator, default constructor, destructor, and copy constructor.

Overriding Built-in Operations

```
#include "String.h"

class Employee {
public:
     Employee();
     Employee(const String &, float);
     Employee(const Employee &);
     ~Employee();

     Employee &operator=(const Employee &);

     void set_name(const String &);
     String get_name() const;

     void set_salary(float);
     float get_salary() const;
     //...

private:
     String name;
     float salary;
};
```

Defining the New Employee Functions

If class Employee creates its own constructor and destructor functions, then those functions will be used together with the String constructor and destructor functions during the creating and destruction of Employees. When an Employee is created, the String constructor is used *before* the Employee constructor. When an Employee is destroyed, the String destructor is used *after* the Employee destructor. This ensures that the Employee's name will already be a legal String when the Employee constructor runs, and that it will still be legal while the Employee destructor runs. If this were not true, then Employee constructor and destructor could not use the String member functions on the name, since those member functions may rely on the representation invariant.

If we wish to use a constructor other than the default constructor for the creation of a member, we must provide arguments for that constructor. The arguments for the data members' constructors are listed in the definition of the constructor, before the body of the function, and separated from the constructor's parameter list by a single colon. The initializers for each member are listed in parenthesis, after the member name. For example, the second Employee constructor on the facing page initializes the name member with the value n, and the salary member with the value f. The expressions used to compute the initial values can work with global variables and functions as well as the constructor's arguments. Members of built-in types, like float, as well as those of class types with constructors, can be initialized.

If we do not specify initial values for the data members, then the default constructor for that member (e.g., String::String()) will be used. Note that if the data member's class has one or more constructors, but no default constructor, then we *must* provide arguments for one of those constructors. If class String had no default constructor, we would get an error if we tried to compile an Employee constructor that does not provide arguments to a String constructor, or if class Employee had no constructor.

If Employee had the constructors below, then when an Employee variable was created, the constructor String::String() would have been used to initialize the name to an empty String, and then the Employee constructor would assign a new value to the name.

```
Employee::Employee(const String &n, float f)
{
    set_name(n);
    set_salary(f);
}

Employee::Employee(const Employee &initializer)
{
    name = initializer.name;
    salary = initializer.salary;
}
```

Note that the definitions of the default constructor, copy constructor, destructor, and assignment operator are equivalent to the mechanisms used by C++, and therefore unnecessary in this example. The only function that provides something new for the users of class Employee is the constructor Employee::Employee(const String &n, float f)

Defining the New Employee Functions

```
#include "Employee2.h"

Employee::Employee()
{
}

Employee::Employee(const String &n, float f)
        : name(n), salary(f)
{
}

Employee::Employee(const Employee &initializer)
        : name(initializer.name), salary(initializer.salary)
{
}

Employee::~Employee()
{
}

Employee &Employee::operator=(const Employee &rhs)
{
    name = rhs.name;
    salary = rhs.salary;
    return *this;
}
```

Using class Employee

Now that we have provided a default constructor, copy constructor, destructor, and assignment operator, C++ will use them. The constructor `Employee::Employee(const String &n, float f)` lets users of class `Employee` initialize `Employee` variables with a name and salary.

Using class Employee

```
#include "Employee2.h"

main(int, char *[])
{
     String jones = "Jones";

     Employee e1(jones, 30000); // String constructor and
              // Employee::Employee(const String &, float)

     Employee e2;              // String constructor and
                               // Employee::Employee()

     e2 = e1;            // Employee::operator=(const Employee &)
                           // (which calls String::operator=)
     Employee e3 = e2; // String constructor and
                      // Employee::Employee(const Employee &)

     return 0;        // Employee::~Employee() and
                      // String::~String()
}
```

Summary

Summary

The built-in definitions for:

- default constructor
- copy constructor
- destructor
- assignment operator

will handle class-type members correctly.

They can be overridden.

Class-type Members

Lab Exercises

UNIT 8

Lab Exercises

1. Change to the *unit08/point* directory. Create a constructor for class **Point** that will allow the initialization of **Point** variables with pairs of integer values. Even though it may not do anything, add a destructor for the class **Point**. The implementation for the constructor and destructor should be added to the file *point.c* and the *Point.h* header file should be updated. A copy of the program *use_point.c* from the *unit07/point* directory has been placed in your *unit08/point* directory. Use it to test your new class **Point**.

   You can compile and execute this program by entering '**make**' or you can compile and execute it directly using the commands:

```
$ CC -o use_point use_point.c point.c print.c
$ use_point
```

SUMMARY	
DIRECTORY	unit08/point
DECLARATION	Point.h (modify), print.h
IMPLEMENTATION	point.c (modify), print.c
TEST PROGRAM	use_point.c

FILE: **use_point.c**

```
#include "Point.h"
#include "print.h"
#include <stdio.h>

main(int, char *[])
{
        Point a;
        Point b(10,10);
        const Point c(100, 100);

        printf(" initially, a, b, and c are:\n");
        print(a);
        print(b);
        print(c);

        a = b + c;
        printf(" after \"a = b + c;\"\n");
        print(a);
        print(b);
        print(c);

        return 0;
}
```

2. Change to the *unit08/string* directory. The goal of this exercise is to determine when constructor and destructor functions will be called. Add a call to **printf** to each constructor, the destructor, and the assignment operator of the class **String**. These functions can be found in the file *encap.c*. The **printf** function calls should print the name of the function being called, and its argument types.

   Try to predict when the constructors and destructor will be called in each test program (*lab2a.c* through *lab2f.c*), and test your prediction by compiling and running the program.

   You can compile and execute each test program by entering '**make prob2***n*' (*n* = a, b, c, d, e, or f); you can compile and execute all of them by entering '**make**'; or, you can compile and execute them directly using the commands:

```
$ CC lab2a.c string.c encap.c -o lab2a
$ lab2a
$ CC lab2b.c string.c encap.c -o lab2b
$ lab2b
$ CC lab2c.c string.c encap.c -o lab2c
$ lab2c
$ CC lab2d.c string.c encap.c -o lab2d
$ lab2d
$ CC lab2e.c string.c encap.c -o lab2e
$ lab2e
$ CC lab2f.c string.c encap.c -o lab2f
$ lab2f
```

SUMMARY	
DIRECTORY	unit08/string
DECLARATION	String.h
IMPLEMENTATION	string.c, encap.c (modify)
TEST PROGRAM	lab2*x*.c (x = a, b, c, d, e, f)

FILE: **lab2a.c**

```
#include "String.h"

main() // test 1
{
     String s1 = "hello\n";
     s1.print();
     return 0;
}
```

FILE: **lab2b.c**

```
#include "String.h"

main() // test 2
{
     String s1;
     s1 = "hello\n";
     s1.print();
     return 0;
}
```

FILE: **lab2c.c**

```
#include "String.h"

main() // test 3
{
     String s1 = "hello\n";
     String s2 = s1;
     s1.print();
     s2.print();
     return 0;
}
```

FILE: **lab2d.c**

```
#include "String.h"

main() // test 4
{
     String s1 = "hello\n", s2;
     s2 = s1;
     s1.print();
     s2.print();
     return 0;
}
```

FILE: **lab2e.c**

```
#include "String.h"

void do_nothing()
{
                     String tmp = "useless local variable\n";
                     tmp.print();
}

main() // test 5
{
     String s1 = "hello\n", s2;
     do_nothing();
     s2 = s1;
     s1.print();
     s2.print();
     return 0;
}
```

FILE: **lab2f.c**

```
#include "String.h"

String &do_nothing(String &str)
{
                    String tmp = "useless local variable\n";
                    tmp.print();

                    return str;
}

main() // test 5
{
     String s1 = "hello\n", s2;
     s2 = do_nothing(s1);
     s1.print();
     s2.print();
     return 0;
}
```

3. Change to the *unit08/employee* directory. Compile the file *use_empl.c* in this directory. It uses the class **Employee** that is defined in *Employee1.h,* shown on page 8-33 of the Student Guide.

   You can compile and execute this program by entering **'make prob3'** or you can compile and execute it directly using the commands:

   ```
   $ CC use_empl.c string.c encap.c -o use_empl
   $ use_empl
   ```

   Class **Employee** will now use the **String** member functions that were modified in the Unit 8 lab exercises. Predict the output of the program *use_empl.c*, and check your results.

SUMMARY	
DIRECTORY	unit08/employee
DECLARATION	Employee1.h, String.h
IMPLEMENTATION	string.c, encap.c
TEST PROGRAM	use_empl.c

FILE: **`use_emp1.c`**

```
#include "Employee1.h"

main(int, char *[])
{
     String jones = "Jones";

     Employee e1, e2;

     e1.set_name(jones);
     e1.set_salary(30000);

     e2 = e1;

     Employee e3 = e2;

     return 0;
}
```

4. Remain in the *unit08/employee* directory. Repeat the previous exercise with the file *use_emp2.c*, which uses the class **Employee** defined in *Employee2.h* (shown on page 8-37 of the Student Guide). The member functions for this class **Employee** are in *employee2.c* (which is shown on page 8-39 of the Student Guide). Predict the output of the program *use_emp2.c*, and check your results.

   You can compile and execute this program by entering '**make prob4**' or you can compile and execute it directly using the commands:

   ```
   $ CC use_emp2.c employee2.c string.c encap.c -o \
        use_emp2
   $ use_emp2
   ```

SUMMARY	
DIRECTORY	unit08/employee
DECLARATION	Employee2.h, String.h
IMPLEMENTATION	employee2.c, string.c, encap.c
TEST PROGRAM	use_emp2.c

FILE: **`use_emp2.c`**

```
#include "Employee2.h"

main(int, char *[])
{
     String jones = "Jones";

     Employee e1(jones, 30000);
```

```
        Employee e2;

        e2 = e1;

        Employee e3 = e2;

        return 0;
    }
```

---

5. Remain in the *unit08/employee* directory. The file *emp2_var.c* is a copy of the file *employee2.c*. Change the constructors in this file (*emp2_var.c*) so that they do not use the member initialization syntax. The file on page 8-38 of the Student Guide shows how this can be done, so you can simply replace the constructor functions in *emp2_var.c* with those shown on page 8-38. Recompile and execute the same test program *use_emp2.c* used in the previous exercise. Predict the output of the program, and check your results.

   You can compile and execute this program by entering '**make prob5**' or you can compile and execute it directly using the commands:

```
$ CC use_emp2.c emp2_var.c string.c encap.c -o \
       use_emp2_var
$ use_emp2
```

SUMMARY	
DIRECTORY	unit08/employee
DECLARATION	Employee2.h, String.h
IMPLEMENTATION	emp2_var.c (modify), string.c, encap.c
TEST PROGRAM	use_emp2.c

UNIT 8

Lab Exercises (Answers)

1. The constructor **Point::Point(int, int)** will allow the initialization of **Points** with pairs of **int** values as in the declaration "Point b(10, 10);". If we still wish to allow the declaration of uninitialized **Points** (e.g., "Point a;"), we must also provide a constructor with no arguments: **Point::Point()**.

FILE: **Point.h**

```
class Point {
public:
        Point(int x, int y);
        Point();
        ~Point();

        int x() const;
        int y() const;
        void set_to(int x, int y);

        Point operator+(const Point &) const;
        Point operator-(const Point &) const;

        friend Point operator*(int,  const Point &);
        friend Point operator*(const Point &, int);

        Point operator/(int) const;

        int operator==(const Point &) const;
        int operator!=(const Point &) const;

        Point &operator+=(const Point &);

private:
        int _x;
        int _y;
};


inline int Point::x() const
{
   return (_x);
}

inline int Point::y() const
{
   return(_y);
}
```

FILE: point.c

```
#include "Point.h"
#include <stdio.h>

void Point::set_to(int x, int y)
{
   _x = x;
   _y = y;
}

Point Point::operator+(const Point &p) const
{
   Point temp;
   temp._x = _x + p._x;
   temp._y = _y + p._y;
   return temp;
}

Point Point::operator-(const Point &p) const
{
   Point temp;
   temp._x = _x - p._x;
   temp._y = _y - p._y;
   return temp;
}

Point operator*(int i, const Point &p)
{
   Point temp;
   temp._x = i * p._x;
   temp._y = i * p._y;
   return temp;
}

Point operator*(const Point &p, int i)
{
   Point temp;
   temp._x = i * p._x;
   temp._y = i * p._y;
   return temp;
}

Point Point::operator/(int i) const
{
   Point temp;
   temp._x = _x / i;
   temp._y = _y / i;
   return temp;
}

int Point::operator==(const Point &p) const
{
   return (_x == p._x && _y == p._y)?1:0;
}


int Point::operator!=(const Point &p) const
{
   return (_x != p._x || _y != p._y)?1:0;
}

Point &Point::operator+=(const Point &p)
{
   *this = *this + p;
   return *this;
}

Point::Point(int x, int y)
{
   _x = x;
   _y = y;
}

Point::Point()
{
}

Point::~Point()
{
}
```

2. The **constructor**, **destructor**, and **operator=** functions can be modified as follows:

FILE: **encap.c**

```
#include "String.h"
#include <string.h>
#include <stdio.h>

//
// constructor function
// called automatically when a string is created.
//
// ensure the string is null terminated
//

String::String()
{
     printf("called String::String()\n");
     text[0] = '\0';
}

String::String(const char *init)
{
     printf("called String::String(const char *)\n");
     printf("\twith argument %s\n", init);
     strncpy(text, init, max_string_length);
     text[max_string_length] = '\0';
}

//
// constructor function
// called automatically when a string is created
// and initialized with a String value
//

String::String(const String &init)
```

```
{
     printf("called String::String(const String &)\n");
     printf("\twith argument %s\n", init.text);
     strcpy(text, init.text);
}

// use: s = "hello world"

String &String::operator=(const char *rhs)
{
     printf("called String::operator=(const char *)\n");
     printf("\twith this = %s\n", text);
     printf("\tand argument %s\n", rhs);
     strncpy(text, rhs, max_string_length);
     text[max_string_length] = '\0';

     return *this;
}

//
// destructor function
// called automatically when a string is destroyed.
//

String::~String()
{
     printf("called String::~String() for string: %s\n", text);
     // nothing needed here yet
}

//
// constructor function
// called automatically when a string is created
// and initialized with a char* value
//

// use: s = t

String &String::operator=(const String &rhs)
{
     printf("called String::operator=(const String &)\n");
     printf("\twith this = %s\n", text);
     printf("\tand argument %s\n", rhs.text);
     strcpy(text, rhs.text);

     return *this;
}
```

5 The new constructors for the class **Employee are:**

FILE: emp2_var.c

```
#include "Employee2.h"

Employee::Employee()
{
}

Employee::Employee(const String &n, float f)
```

```
{
     set_name(n);
     set_salary(f);
}

Employee::Employee(const Employee &initializer)
{
     name = initializer.name;
     salary = initializer.salary;
}

Employee::~Employee()
{
}

Employee &Employee::operator=(const Employee &rhs)
{
     name = rhs.name;
     salary = rhs.salary;
     return *this;
}
```