```
}

int operator!=(String &s1, String &s2)
{
   return strcmp(s1.text, s2.text);
}


int operator==(String &s1, char *s2)
{
   return !strcmp(s1.text, s2);
}

int operator!=(String &s1, char *s2)
{
   return strcmp(s1.text, s2);
}

int operator==(char *s1, String &s2)
{
   return !strcmp(s2.text, s1);
}

int operator!=(char *s1, String &s2)
{
   return strcmp(s2.text, s1);
}
```

```
   return (_x != p._x || _y != p._y)?1:0;
}

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

3. If we change the comparison operators to use call by reference, they will run more efficiently, since they do not need to make copies of their arguments. The test programs must be re-compiled, but we do not have to change the source code.

FILE: **String.h**

```
const int max_string_length = 128;

class String {




   return (_x != p._x || _y != p._y)?1:0;
}
```

2. The **operator+=** has a reference parameter, corresponding to the **Point** on the right hand side of an expression like "a += b". The **operator+=** function will not change its parameter, so it will not affect the right hand operand. The left hand operand will become the invoking object of **operator+=**. When **operator+=** changes its invoking object, it will affect the left hand side of the assignment. To return the value assigned to the left hand operand, **operator+=** can simply return the value of its invoking object (***this**). Since the invoking object will still exist after the call to **operator+=**, **operator+=** can return a reference to its invoking object.

FILE: **Point.h**

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

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

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

       Point operator/(int);

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

       Point &operator+=(Point &);

private:
       int _x;
       int _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+(Point &p)
{
   Point temp;
   temp._x = _x + p._x;
   temp._y = _y + p._y;
   return temp;
}

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

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

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

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

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


int Point::operator!=(Point &p)
{
```

UNIT 6

Lab Exercises (Answers)

1. All the functions that have **Point** arguments, and do not change those arguments, would be more efficient with call by reference. The **int** arguments do not need to be passed by reference, as copying an **int** is just as fast as passing a reference to an **int**. Arguments that are changed by the function should not be passed by reference unless the function is supposed to change the actual argument passed by the calling function.

   Since all the functions of class **Point** return local variables, none should return a reference.

FILE: **Point.h**

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

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

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

        Point operator/(int);

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

private:
        int _x;
        int _y;
};


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

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

SUMMARY	
DIRECTORY	unit06/string
DECLARATION	String.h (modify)
IMPLEMENTATION	same.c (modify), string.c
TEST PROGRAM	str_same.c

FILE: **use_point.c**

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

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

        a.set_to(1, 1);
        b.set_to(10, 10);
        c.set_to(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);

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

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

        return 0;
}
```

3. Change to the *unit06/string* directory. Change your **operator!=** and **operator==** functions for class **String** to use call by reference. After making the changes, use the *str_same.c* program to test your new version. This program is a copy of the one you used in the Unit 5 Lab Exercises. Do you need to change the test program? You can compile and execute this program by entering '**make prob3**' or you can compile and execute it directly using the commands:

```
$ CC -o str_same str_same.c string.c same.c
$ str_same
```

UNIT 6

Lab Exercises

1. Change to the *unit06/point* directory. Decide which functions in your class **Point** would benefit from the use of call by reference, and change them to use call by reference. Should any of them return a reference?

   After making the changes, use the *use_ref.c* program to test your new version. You can compile and execute this program by entering '**make prob1**' or you can compile and execute it directly using the commands:

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

SUMMARY	
DIRECTORY	unit06/point
DECLARATION	Point.h (modify), print.h
IMPLEMENTATION	point.c (modify), print.c
TEST PROGRAM	use_ref.c

2. Your class **Point** includes a + operator to add two Points. Overload the **operator+=**, and test it with the program *use_point.c*. Notice that the statement "a = b + c" should change only **a**, the statement "a += b" should change **a**, and that "a = b += c" should change **a** and **b**. Should **operator+=** have a reference argument? Should it return a reference?

   You can compile and execute the test program by entering '**make prob2**' 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	unit06/point
DECLARATION	Point.h (modify), print.h
IMPLEMENTATION	point.c (modify), print.c
TEST PROGRAM	use_point.c

Lab Exercises

References

Summary

- Call by reference
  - — Looks like call by value
  - — Efficiency and side effects of passing a pointer
- Returning a reference
  - — Allows the use of function call as an lvalue

Summary

Using the Subscript Operator

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

main(int, char *[])
{
    String s;
    int i;
    char ch;

    printf("enter a string: ");
    s.read();

    for (i=0; i<s.length(); i++) {
      ch = s[i];
      s[i] = toupper(ch);
      // or, s[i] = toupper(s[i]);
    }

    s.print();

    return 0;
}
```

Using the Subscript Operator

Since the `operator[]` returns a reference, we can use it on the left side of an assignment (it is an *lvalue*). Note that the statement `s[i] = toupper(ch)` assigns a character value to a character in the `String`. Since this is an assignment of one character to another, it does not require a user-defined `operator=`.

Defining a Subscript Operator

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

//
// operator[] for Strings -- returns a reference
//  to the appropriate character in the String
//
//      s[index]
//  is equivalent to
//      s .operator[]( index )
//  which is another name for
//      s.text[index]
//  or an error if index is invalid
//

char &String::operator [](int index)
{
    if (index < 0 || index >= length()) {
      fprintf(stderr,
          "Illegal index (%d) for String \"%s\".\n",
          index, text);
      exit(1);
    }

    return text[index];
}
```

Defining a Subscript Operator

The subscript operator finds the appropriate character within character array of the String. If it returned a value, that character would be *copied* back to the calling function, which would not be able to change the text of the String. Since it returns a reference, the function call is treated as another name for the character returned, so the calling function can change that character.

The Subscript Operator

```
const int max_string_length = 128;

class String {

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

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

     friend String operator+(String &, String &);
     friend String operator+(String &, char *);
     friend String operator+(char *, 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];
};
```

The Subscript Operator

Another useful `String` operation is subscripting. We would like to allow the use of the subscript operator on either the left or right side of an assignment. This is not legal if it returns a copy of the character, since that copy will be stored in compiler generated temporary space (i.e. it is not an *lvalue).* If, on the other hand, `operator[]` returns a reference to the character, we can use the subscript operator to change the character within a `String`.

String::operator=

```
#include "String2.h"
#include <string.h>

// use: s = "hello world"

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

    return *this;
}
```

String::operator=

The only change to the definition of operator= is the return type. operator= can still be used in the same way:

```
#include "String2.h"

main (int, char *[])
{
    String first, last;

    first = "Zaphod";
    last = "Beeblebrox";
    // both of the above call
    // String::operator=(char *)

    return 0;
}
```

Returning a Reference from operator=

```
const int max_string_length = 128;

class String {

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

     String substring(int start, int len);

     friend String operator+(String &, String &);
     friend String operator+(String &, char *);
     friend String operator+(char *, 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];
};
```

Returning a Reference from operator=

We can improve the efficiency of our `operator=` by returning a reference. Since the returned `String` is no longer copied, the return from the function will be faster.

Returning a Reference

Returning a reference

- return value is not copied back
- calling function can change returned object
- may be faster than returning a value
- can not be used with local variables

```
int &max(int &i, int &j)
{
    if (i > j)
      return i;
    else
      return j;
}

main(int, char *[])
{
    int x = 42, y=7500000, z;

    z = max(x, y);
    // z is now 7500000

    max(x, y) = 1;
    // y is now 1

    return 0;
}
```

Returning a Reference

Using Operators with Reference Arguments

```
#include "String.h"

main (int, char *[])
{
    String first, last, full;

    first = "Zaphod";
    last = "Beeblebrox";
    // both of the above call
    // String::operator=(char *)

    full = first + " " + last;
    // calls operator+(String &, char *)
    // and   operator+(String &, String &)

    ("Name is:  " + full).print();
    // calls operator+(char *, String &)
    // and   prints the result

    return 0;
}
```

Using Operators with Reference Arguments

At last, we can use the "+" symbol to denote concatenation of `Strings`, without the inefficiency of an unnecessary call by value. We can add `Strings` to each other, or to groups of quoted characters. Now that these operations use call by reference, they are just as fast as the original `concat` functions.

Defining operator +

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

String operator+(String &lhs, String &rhs)
{
    String both;

    if (lhs.length() + rhs.length() > max_string_length) {
      fprintf(stderr, "RUN TIME ERROR: String too large\n");
      exit(1);
    }
    strcpy(both.text, lhs.text);
    strcat(both.text, rhs.text);
    return both;
}

String operator+(char *lhs, String &rhs)
{
    String both;

    if (strlen(lhs) + rhs.length() > max_string_length) {
      fprintf(stderr, "RUN TIME ERROR: String too large\n");
      exit(1);
    }
    strcpy(both.text, lhs);
    strcat(both.text, rhs.text);
    return both;
}
```

Defining operator +

The declaration of a function must match its definition, so we must change the declaration of the addition operations in class String to show that they use call by reference.

Now the `operator+` functions will not have the overhead of copying their `String` arguments. Note that there is no need to change the `char *` arguments to use call by reference, since call by reference is no more efficient than passing a pointer.

Since none of the `operator+` functions made any changes to `lhs` or `rhs`, the change from call by value to call by reference will not "break" code that uses our `operator+` functions (although such code must be re-compiled). If an `operator+` function had changed `lhs` or `rhs`, that function would now cause a change to a calling function's variable, and possibly affect that function.

operator + with Call by Reference

```
const int max_string_length = 128;

class String {

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

     String substring(int start, int len);

   friend String operator+(String &, String &);
   friend String operator+(String &, char *);
   friend String operator+(char *, 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];
};
```

operator + with Call by Reference

If we change the `operator+` functions so that their parameters are references to `Strings`, the arguments will not be copied when the function is called. Since these functions do not change their arguments, this change will effect only the efficiency, and not the result, of programs using `operator+`.

References

A reference is

- an additional name for existing object
- declared *type &name*
- often used for function parameters:
  - — called function can change actual argument
  - — faster than call by value for large objects

```
void set_to_five(int &i)
{
    i = 5;
}

main(int, char *[])
{
    int number = 78;

    set_to_five(number);
    // number is now 5.

    return 0;
}
```

Call by Reference

In C, when a variable is passed as an argument to a function, the function's parameter becomes a copy of the value passed. Any use of the parameter name inside the called function will refer to the copy, and the called function can change this copy without affecting the variable of the calling function. This mechanism is known as *call by value*, and it is used for all function arguments in C, including pointers (in which case the pointer, but not the storage pointed to, is copied). If the function on the facing page used call by value, `i` would become a name for a *copy* of `main's` variable `number`. Changes to `i` would not affect `number`.

In C++, you can declare a parameter that is a *reference*, to the original value. In this case, the original value is *not* copied. Any use of the parameter name inside the called function will refer to the *original* value in the calling function. This mechanism allows the called function to change the variable passed by the calling function, and it can reduce the storage and time needed to call the function (since there is no need to copy the argument). On the facing page, `i` is a name for `number` (*not* a copy of `number`), so `set_to_five` changes the value of `number` to 5.

To perform call by reference, C++ passes a pointer, and dereferences that pointer inside the called function, to refer back to the original value in the called function. Call by reference combines the efficiency of passing a pointer and the ability to change the calling function's variables with the syntax of call by value.

Since a call by reference looks like a call by value, it is not easy to tell which mechanism is used by examining a function call. This ambiguity is useful in situations in which a you do not need to know which mechanism is being used (for example, when a reference is used only for efficiency). If, on the other hand, you do need to know which mechanism is being used (for example, if a function is designed to change one of its actual arguments), the ambiguity may be confusing. We recommend that call by reference not be used for such functions unless the function name suggests that it will change the calling function's variable.

References can be local, static, or global variables, as well as function parameters:

```
int i;
int &j = i;   // j is another name for i

j = 7;      // now i is 7
```

The code above is legal, and serves to demonstrate that references are a general purpose mechanism that can be used in many situations in C++. The code above does not, however, show a *useful* example of references -- having a local reference to another local variable will probably only make a program more confusing.

Objectives

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

- Pass parameters to a function by reference
- Return references

Objectives

References