FILE: **same.c**

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

int is_the_same_as(String s, String t)
{
   return !strcmp(s.text, t.text);
}

int is_different_from(String s, String t)
{
   return strcmp(s.text, t.text);
}

int is_the_same_as(String s, char *t)
{
   return !strcmp(s.text, t);
}

int is_different_from(String s, char *t)
{
   return strcmp(s.text, t);
}

int is_the_same_as(char *s, String t)
{
   return !strcmp(s, t.text);
}

int is_different_from(char *s, String t)
{
   return strcmp(s, t.text);
}
```

```
void print (double f)
{
  printf("%lg", f);
}

void print (int i)
{
  printf("%d", i);
}

void print(Point p)
{
   printf("(%d,%d)\n", p.x(), p.y());
}
```

It is not slowed down by the calls to the member functions **x** and **y**, however, since these functions are inline functions. We have achieved the efficiency of the friend function that we wrote for question one together with the ease of maintenance that we had before we made **print(Point)** a friend.

3.

FILE: **String.h**

```
const int max_string_length = 128;

class String {

public:
     void set_to(char *);
     int  length();
```

UNIT 4

Lab Exercises (Answers)

1. Class **Point** must now declare **print(Point)** as a friend function:

FILE: **Fr.Point.h**

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

     friend void print(Point);

private:
     int _x, _y;
};
```

Since *Point.h* contains the declaration of **print(Point)**, the file *print_pt.h* is not needed. The code for **print(Point)** can be left in *print_pt.c*, but since it is now a

```
  else
    printf("doesn't work.\n");

  printf("\nis_different_from(String, char *) ");
  if (!is_different_from(h1, "hello") && is_different_from(h1, "world"))
    printf("works.\n");
  else
    printf("doesn't work.\n");

// Test new functions from unit 4

  printf("\nis_the_same_as(char *, String) ");
  if (is_the_same_as("hello", h1) && !is_the_same_as("world", h1))
    printf("works.\n");
  else
    printf("doesn't work.\n");

  printf("\nis_different_from(char *, String ");
  if (!is_different_from("hello", h1) && is_different_from("world", h1))
    printf("works.\n");
  else
    printf("doesn't work.\n");

  return 0;
}
```

3. Change to the *unit04/string* directory. The file *same.c* contains the **is_the_same_as** and **is_different_from** functions resulting from the unit03 exercises. Write two more functions to compare character arrays to **Strings**: **is_the_same_as(char *, String)** and **is_different_from(char *, String)**. These new functions should be friends of class **String**. While you are adding these two new functions to the files *String.h* and *same.c*, convert the existing member functions with those names (that you wrote in the last lab) into friend functions to get more practice with friend functions.

   You can compile and execute the test 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
```

SUMMARY	
DIRECTORY	unit04/string
DECLARATION	String.h (modify)
IMPLEMENTATION	same.c (modify), string.c
TEST PROGRAM	str_same.c

FILE: **str_same.c**

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

main(int, char *[])
{
  String h1, h2, w;

  h1.set_to("hello");
  h2.set_to("hello");
  w.set_to("world");

  printf("\nis_the_same_as(String, String) ");
  if (is_the_same_as(h1, h2) && !is_the_same_as(h1, w))
    printf("works.\n");
  else
    printf("doesn't work.\n");

  printf("\nis_different_from(String, String) ");
  if (!is_different_from(h1, h2) && is_different_from(h1, w))
    printf("works.\n");
  else
    printf("doesn't work.\n");


  printf("\nis_the_same_as(String, char *) ");
  if (is_the_same_as(h1, "hello") && !is_the_same_as(h1, "world"))
    printf("works.\n");
```

UNIT 4

Lab Exercises

1. Change to the *unit04/point* directory. The files *Fr.Point.h* and *Fr.print.c* are copies of the files *Point.h* and *print.c*. For this exercise you are to modify the files *Fr.Point.h* and *Fr.print.c*.

   Make your **print(Point)** function a friend of class **Point**, and change it (remember to use the files *Fr.Point.h* and *Fr.print.c*) so that it accesses the private data members of the **Point** it is printing. How does the new function compare to the original in terms of efficiency? How many functions will need to be re-written if we change the representation of points? How many would have been re-written if we had not made **print(Point)** into a friend of class **Point**?

   You can compile and execute the test program by entering '**make prob1**' or you can compile and execute it directly using the commands:

```
$ CC -o print_test1 print_test1.c point.c Fr.print.c
$ print_test1
```

SUMMARY	
DIRECTORY	unit04/point
DECLARATION	Fr.Point.h (modify), print.h
IMPLEMENTATION	Fr.print.c (modify), point.c
TEST PROGRAM	print_test1.c

2. This exercise uses the files *Point.h*, *print.c*, and *print.h* where the functions are not declared as friends. Make the **x** and **y** functions of class **Point** into inline functions. How does the efficiency of print compare to the original function? To the friend function in question one? How many functions will need to be re-written if we change the representation of points?

   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 print_test2 print_test2.c point.c print.c
$ print_test2
```

SUMMARY	
DIRECTORY	unit04/point
DECLARATION	Point.h (modify), print.h
IMPLEMENTATION	point.c (modify), print.c
TEST PROGRAM	print_test2.c

Lab Exercises

Summary

A friend function can be used instead of a member when:

- the first argument is not of the class type
- a function must access the private section of more than one class

In other situations, use a member function instead.

Summary

We have seen that member functions may be necessary when the first argument of a function is not of the class type, or when a function must access the private section of more than one class. In other situations, member functions should be chosen instead, because dynamic binding can only be used with member functions (we will study dynamic binding later in the course).

Example Inline Functions

```
class Matrix {
public:
    int elem(int i, int j);
    void set_elem(int i, int j, int val);

private:
    int elements[3][3];
};


inline int Matrix::elem(int i, int j)
{
    return elements[i][j];
}

inline void Matrix::set_elem(int i, int j, int val)
{
    elements[i][j] = val;
}
```

Example Inline Functions

We can avoid the overhead of the "extra level" of function calls if we make the member functions that access the data of a matrix into inline functions.

Inline functions can also be created by putting the code for a function into the class declaration:

```
class Matrix {
    int elements[3][3];

public:
    int elem(int i, int j)  { return elements[i][j]; }
    void set_elem(int i, int j, int val) { elements[i][j] = val; }
};
```

This style is not recommended because it puts the code for the function in the public section (a user of the class should not have to read through it when looking at the public section), and it can make the class declarations much longer, and forcing us to put the private section first.

Inline Functions

Inline functions act like functions:

- They can be class members
- Type checking is performed
- They can be overloaded
- They obey normal parameter passing rules

But they are implemented like macros:

- Code is substituted inline, not called
- Use is faster than calling a function
- Use may take more space
- They are defined in .h files, not .c files

Inline Functions

In C, if a small piece of code is used over and over again in parts of a program where execution speed is critical, that code is often written as a macro instead of a function. Macros are still legal in C++, but they do not fit in well with the other C++ features: they can not be listed as friends or members of a class, their arguments' types are not checked, so they can not be overloaded, and the arguments do not obey the normal parameter passing rules (this can be a problem in C, as well).

C++ provides inline function expansion, to make up for these problems with macro expansion. If we move a function into a header file and label it with the word "inline", C++ will expand it inline rather than generating function calls to it, if possible. If the function has any parameters that were passed by value, C++ only copies the value if it will be changed in the function, to avoid unnecessary overhead. Some constructs are too complicated for the C++ inline function mechanism, in which case there will usually be an error or warning from the compiler.

If an inline function is changed, all used of that inline function must be re-compiled, so inline functions should only be used for functions that are not likely to change.

Calling Existing Members

```
class Vector
{
public:
     void set_elem(int, int);
     int elem(int);
     //...
private:
     int elements[3];
};

class Matrix
{
public:
     void set_elem(int, int, int);
     int elem(int, int);
     //...
private:
     int elements[3][3];
};

Vector mult(Matrix *m, Vector *v);
```

Calling Existing Members

Often it is possible to write a function in terms of existing member and friend functions. In this case, it does not need access to the private section, so it does not need to be either a member or a friend. Since it is neither a member or a friend, it is can be written without knowledge of the implementation of either class. This is the usual way to create functions that work with more than one class, but it only works if the classes provide enough members and friends. The `mult` function might look like this:

```
#include "Vec_Matr4.h"
// use: mult(m, v);

Vector mult(Matrix *m, Vector *v)
{
    Vector r;
    int i, j;

    for (i=0; i<3; i++) {
       r.set_elem(i, 0);
       for (j=0; j<3; j++)
           r.set_elem(i, r.elem(i) + m->elem(i,j) * v->elem(j));
    }
    return r;
}

void Matrix::set_elem(int i, int j, int val)
{
    elements[i][j] = val;
}
int Matrix::elem(int i, int j)
{
    return elements[i][j];
}

void Vector::set_elem(int i, int val)
{
    elements[i] = val;
}
int Vector::elem(int i)
{
    return elements[i];
}
```

All Members of Another class as Friends

```
class Vector;  // forward reference
               // to make declaration of
               // "mult" legal

class Matrix
{
public:
     Vector mult(Vector *v);
     //...
private:
     int elements[3][3];
};

class Vector
{
public:
     friend class Matrix;
     //...
private:
     int elements[3];
};
```

All Members of Another class as Friends

All member functions of a class may be made friends of another class.

This is an alternative to simply listing each member function as a friend, as shown on the previous page.

Member of Another class as a Friend

```
class Vector;  // forward reference
               // to make declaration of
               // "mult" legal

class Matrix
{
public:
     Vector mult(Vector *v);
     //...
private:
     int elements[3][3];
};

class Vector
{
public:
     friend Vector Matrix::mult(Vector *v);
     //...
private:
     int elements[3];
};
```

Member of Another class as a Friend

A member function may be a friend of another class. This is another ways to write a function that works with objects of different types. Since it also allows access to the private sections of more than one class, it should also only be used when designing several classes at the same time.

If we chose to write the `mult` function as a friend of class Vector and a member of class Matrix, it might look like this:

```
#include "Vec_Matr2.h"

// use: m.mult(v);

Vector Matrix::mult(Vector *v)
{
    Vector r;
    int i, j;

    for (i=0; i<3; i++) {
       r.elements[i] = 0;

       for (j=0; j<3; j++)
           r.elements[i] += this->elements[i][j] * v->elements[j];
    }
    return r;
}
```

Friends of More Than One class

```
class Matrix;  // forward reference
               // to make declaration of
               // "mult" legal

class Vector
{
public:
     friend Vector mult(Matrix *m, Vector *v);
     //...
private:
     int elements[3];
};

class Matrix
{
public:
     friend Vector mult(Matrix *m, Vector *v);
     //...
private:
     int elements[3][3];
};
```

Friends of More Than One class

A function may be a friend of more than one class. If you are designing several classes as part of one library, and need to write a function that works with the private data of more than one of those classes, you can create a function that is a friend of those classes. It is not possible to create a function that is a member of more than one class.

Note that this technique should only be used when you are designing several classes at the same time, and understand the private sections of those classes.

If we choose to write the `mult` function as a friend of both classes, it might look like this:

```
#include "Vec_Matr.h"

// use: mult(m, v);

Vector mult(Matrix *m, Vector *v)
{
    Vector r;
    int i, j;

    for (i=0; i<3; i++) {
       r.elements[i] = 0;

       for (j=0; j<3; j++)
           r.elements[i] += m->elements[i][j] * v->elements[j];
    }
    return r;
}
```

Calling Friend Functions

```
#include "String.h"

main(int, char *[])
{
    String firstname, lastname, name, output;

    firstname.set_to("Zaphod");
    lastname.set_to("Beeblebrox");

    name = concat(&firstname, " ");
    name = concat(&name, &lastname);
    output = concat("Name is: ", &name);

    output.print();

    return 0;
}
```

Calling Friend Functions

If we had tried to write the third `concat` function as a member, it would have required an invoking object of type `String`, and would have looked something like the function `reverse_concat` shown below:

```
#include "String.h"

main(int, char *[])
{
   String firstname, lastname, name, output;

   firstname.set_to("Zaphod");
   lastname.set_to("Beeblebrox");

   name = firstname.concat(" ");
   name = name.concat(&lastname);
   output = name.reverse_concat("Name is: ");

   output.print();

   return 0;
}
```

Writing Friend Functions

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

String concat(String *first, String *other)
{
    String both;

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

String concat(char *first, String *second)
{
    String both;

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

Writing Friend Functions

Friend functions are written just like other functions that are not member functions of the class, but they can access the private section of object of the class.

Declaring Friend Functions

```
const int max_string_length = 128;

class String {

public:
     void set_to(char *);
     int  length();
     int  read();
     void print();

     String substring(int start, int len);

   friend String concat(String *, String *);
   friend String concat(String *, char *);
   friend String concat(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];
};
```

Declaring Friend Functions

Friend functions are distinguished from members with the keyword **friend.** Since friend functions do not have invoking objects, each `concat` function must take two arguments. Note that we have added a third `concat` function, which concatenates a character array and a `String`.

The distinction between the public and private sections of a class applies only to members of the class. A friend function may be declared in either the public or the private section of a class, without any effect on its scope. We recommend placing friend functions in the public section of a class, so that users need only look at the public section to see all the defining operations.

Friend Functions

Member functions

- are part of one class
- can access the class's private section
- are called with an "invoking object" of that class type
- can use
  - — the keyword **this**
  - — the names of members

  to refer to the invoking object

Friend functions

- are declared in one or more classes
- have access to the private sections of those classes
- are distinguished from members with the keyword **friend**
- are not called with an invoking object of those classes

Friend Functions

In C++, there are two kinds of defining operations for a class: member functions and friend functions. Both must be listed in the class declaration, so both have access to the class's private section.

Only member functions are invoked by objects, so only member functions can use the keyword **this** or the member names to refer to the invoking object. If a friend function needs to work with some object of the class, that object must be passed as an argument.

Objectives

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

- Declare, define and use friend functions
- Describe how friends differ from member functions
- Decide whether a function should be written as a member or friend
- Create an inline function

Objectives

Friend Functions