Unit 2

Object-Oriented Programming in C++

Classes and Member Functions

Objectives

Objectives

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

- Define a type in terms of a list of operations
- Choose an implementation for the type
- Write the code for the type's defining operations

Creating a Type

The keyword **class** is used to declare a new type in C++. A class is divided into two sections: the public and the private. The public section lists all the information that will be available to programmers who declare variables of that type, including the list of operations that can be used to manipulate variables of that type. The private section lists information that can only be used inside the class's defining operations.

We will look first at the public section and the use of a class, and then at the private section and the implementation.

Creating a Type

A data abstraction is created with the **class** keyword.

A class declaration is like a struct, except:

- it lists defining operations as well as data
- members may be public or private

A class is used to declare variables.

Member function call syntax:

variable-name.member-function(arguments)

A Simple class String

Our class `String` has only four defining operations:

- set_to, to give a `String` a value
- length, to find the length of a `String`
- read, to read in a `String`
- print, to print out a `String`

Since these functions are listed as members of the class (like members of a structure in C), they are known as the "member functions" of class `String`. All members declared in a class are part of the class, and their names will not be confused with names outside the class (like the names of members of a **struct** in C). Member names are said to be in the class's *scope*.

Note that class `String` is distinct from a quoted group of characters (e.g. "hello, world"), which has type `char *` but are often called strings.

A Simple class String

```
class String {

public:
    void set_to(char *);
    int  length();
    int  read();    // read from stdin
    void print();   // print to stdout

private:
    // implementation details
};
```

Using class String

String variables are declared just like any other variables. They can be manipulated by calling the defining operations listed in the public section of class String. Member functions are called with a special kind of function call, in which the name of the object comes before the function name. We will call the object that invokes the member function the *invoking object.*

```
    arg.length();   // invoking object = arg
    input.length();      // invoking object = input
```

A programmer using class String can write functions that work with String variables, but those functions must be written in terms of the defining operations or other functions (they can not access the implementation of the String directly).

```
class String {

public:
    void set_to(char *);
    int  length();
    int  read();   // read from stdin
    void print();  // print to stdout

private:
    // implementation details
};
```

Using class String

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

main(int argc, char *argv[])
{
    String arg, input;

    if (argc == 2) {
      arg.set_to(argv[1]);
      printf("argument length is %d\n", arg.length());
    }

    printf("enter a string:\n");
    input.read();
    printf("length is %d\n", input.length());

    return 0;
}
```

A Simple Implementation

Before we can compile the test program, we must fill in the private section of class String, and write the code for the member functions. The first step in writing a class is getting the right set of operations, so we will use the a simple implementation of class String in our first tests. We can concentrate on a flexible, efficient implementation of the class once we have decided on the right set of member functions.

During the development of class String, we will use an array of 129 characters to store the text for every String variable, so we declare an array of 129 characters in the private section of class String. Class String is like a "struct" in the C language because it shows what storage will be needed when String variables are created. It is different from a C "struct" because only the String operations can access the data of a String.

Any members declared before the first "public:" or "private:" will be private. In early versions of C++, there was no keyword "private:", so the only way to declare private members was to list them before the keyword "public:". Class declarations written in early versions of C++ look like this: (they can still be compiled, but the style on the facing page is recommended).

```
const int max_string_length = 128;

class String {

    // 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];

public:
    void set_to(char *);
    int  length();
    int  read();   // read from stdin
    void print();  // print to stdout
};
```

A Simple Implementation

```
const int max_string_length = 128;

class String {

public:
    void set_to(char *);
    int  length();
    int  read();    // read from stdin
    void print();   // print to stdout

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];
};
```

Writing Member Functions

The name of any class member may be preceded by its class's name using the *scope resolution operator*. For example, because length is a member function of class `String`, the expression

```
input.length()
```

could be written:

```
input.String::length()
```

The scope resolution operator is not needed for calls to member functions, because C++ will automatically choose member functions from the invoking object's class. The scope resolution operator is required in the definition of a member function, since there is no other way for C++ to tell which class's member function is being defined.

A member function can work with its invoking object by using the keyword **this.** The keyword **this** is implicitly defined as a pointer to the function's invoking object:

```
input.length() // inside String::length, "this" points to input
arg.length()   // inside String::length, "this" points to arg
```

A member function can refer to the individual members of its invoking object by using the member names without specifying an object. For example, a `String` member function can get at its invoking object's `text` with either `this->text` or just `text`.

```
input.length() // "text" or "this->text" means input.text
arg.length()   // "text" or "this->text" means arg.text
```

Writing Member Functions

Defined with *scope resolution operator,* " :: "

```
class-name::member-function-name(parameter-list)
{
    // body
}
```

Member function can refer to:

- its arguments (by using the parameter names)
- its invoking object
  - by using the keyword **this**
  - by using member names

String Member Functions

The `length` member function uses the C library function `strlen` to find number of characters before the null byte in the invoking object's `text` array. It can refer to the invoking objects `text` array with either the expression `text` or the expression `this->text`. In the example `input.length()`, either of the above expressions would refer to `input.text`

`set_to` copies the characters from the argument (of type char *) into the `text` array of the invoking object (a `String`) with `strncpy`. It then stores a null byte in the last element of the array, to ensure that the array will be null terminated even if `s` contains more than `max_string_length` characters (in which case, `strncpy` would not have provided the terminating null.

The `read` and `print` member functions use the stdio library functions `fgets` and `fputs` to input or output a line of characters. `read` also removes the newline character from the end of the input. It can find the length of its invoking object with the expression `length()` or `this->length()`.

Class `String` is shown here for reference:

```
const int max_string_length = 128;

class String {

public:
    void set_to(char *);
    int  length();
    int  read();   // read from stdin
    void print();  // print to stdout

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 Member Functions

```
#include "String.h"
#include <string.h>  // (for standard C string functions)
#include <stdio.h>   // (stdio is used in read, print)

int String::length()
{
    return strlen(text);
    // or, return strlen(this->text);
}

void String::set_to(char *s)
{
    strncpy(text, s, max_string_length);
    text[max_string_length] = 0;
}

int String::read()
{
    if (fgets(text, max_string_length+1, stdin) == 0)
      return 0;  // end of file

    if (text[length()-1] == '\n')
      text[length()-1] = '\0';
    return 1;
}

void String::print()
{
    fputs(text, stdout);
}
```

More String Functions

Any additional work with `String` variables will quickly show that we need more defining operations. For example, we may want to concatenate `Strings`, or find a substring of some `String`.

Note that the concat and substring operations will return `Strings`, and that `Strings` can be assigned. They can also be passed as arguments to functions. This is legal because structure variables could be assigned, returned, or passed as arguments in C.

More String Functions

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

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

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

    name = firstname.concat(&lastname);
    printf("name is:  ");
    name.print();

    firstname = name.substring(0, 6);
    printf("\nfirstname is:  ");
    firstname.print();

    return 0;
}
```

Adding String Functions

The `String::substring` function takes only two arguments, the start position and length of the substring. It does not need to have a `String` argument to know which `String` to use, because it takes a substring of the invoking object.

The concat function returns the concatenation of the invoking `String` and the argument `String`.

Adding String 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);
    // return substring of invoking string,
    //  without changing original.

    String concat(String *);
    // return the concatenation of invoking string
    //  and the argument, without changing either.

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::substring

When the `String::substring` function refers to "text", it means the member `text` of the invoking object. It can use `sub.text` to refer to the `text` of its local variable `sub`.

```
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);
    // return substring of invoking string,
    //  without changing original.

    String concat(String *);
    // return the concatenation of invoking string
    //  and the argument, without changing either.

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::substring

```
#include "String.h"
#include <string.h>
#include <stdio.h>
#include <stdlib.h>  // (for "exit")

String String::substring(int start, int len)
{
     if (start+len >= length() || start < 0) {
       fprintf(stderr,
            "illegal index (%d) for String \"%s\".\n",
            start, text);
       exit(1);
     }
     else {
       String sub;
       int i=0;

       while (i<len) {
            sub.text[i] = text[start+i];
            i++;
       }
       sub.text[i] = '\0';

       return sub;
     }
}
```

String::concat

The `String::concat` function copies the text of the invoking `String` into the temporary `both`, appends the text of the argument `String` after it, and returns `both`.

```
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);
    // return substring of invoking string,
    //  without changing original.

    String concat(String *);
    // return the concatenation of invoking string
    //  and the argument, without changing either.

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::concat

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

String String::concat(String *other)
{
    String both;

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

    return both;
}
```

Summary

Summary

The keyword **class** is used to define a type

- The member functions listed in the public section are the defining operations
- The implementation details are listed in the private section
- Only the class's defining operations can access the private members
- The member functions refer to the invoking object using **this** or member names

Lab Exercises

Unit 2 Lab Exercises

In many units, you will find that there are more lab exercises than you can do in the allotted time (this ensures that even a quick programmer will have something to do). You should therefore decide which questions are most interesting to you, and work on them first. Do not worry if you don't have time to answer all the questions.

Some lab exercises let you create a new class, named `Point`, and others let you modify the class `String` discussed in the lecture. In later units, lab exercises involving class `Point` will depend on your answers to the `Point` questions in this unit, and future lab exercises for class `String` will depend on your answers to the `String` labs in this unit. Therefore, you may find it helpful to pick one of those two classes, and, in each unit, do the labs for that class first.

In this unit, question 1 lets you compile a C++ program. Start with question 1 regardless of which class interests you more. Questions 2 and 3 let you start class `Point`, and question 4 asks you to enhance the `String` class discussed in the lecture.

1) The `String` class discussed in the lecture is in the unit02 directory. The class declaration is in the file `String.h`, and the member functions in the files `simple_fns.c`, `substring.c`, and `concat.c`. Compile and run the program `use_string.c`, which is a sample use of class `String`.

--------------------- FILE: use_string.c

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

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

   s.set_to("hello world");
   printf("Length of s is %d\n", s.length() );

   printf("Enter string t:  ");
   t.read();
   printf("Length of t is %d\n", t.length() );
   return 0;
}
```

----

2) Create a directory named `point` below your home directory, and keep the files for questions 2 and 3 in that directory. This will allow you to return to these exercises at later times during the course. Create a class `Point` that represents a pair of co-ordinates. Such a class might be used to identify positions on the terminal screen in a graphics program. Your class needs only three member functions: one (called `x`) to find the distance along the X axis, one (called `y`) to find the distance along the Y axis, and one, called `set_to`, to give values to the Point's X and Y co-ordinates. (The function's first argument should be the X value, and the second argument the Y.) You may assume that the X and Y values will be integers. You may use whatever private data members seem appropriate (such as two integers named `_x` and `_y`. Note that we should not give the data the same name as the member functions: adding underscores, or capitalizing the names, are ways to prevent this).

Put the class declaration in the file `Point.h`, and the source for the member functions in `point.c`, both in the `point` directory. Test your class with the test program `use_point.c`, which is located in the unit02/Labs directory.

-------------------- FILE: Labs/use_point.c

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

main(int, char *[])
{
  Point p1, p2;

  p1.set_to(3, 5);
  p2.set_to(2, 7);

  if (p1.x() == 3 &&
      p1.y() == 5 &&
      p2.x() == 2 &&
      p2.y() == 7)
  {
    printf("test successful.\n");
    return 0;
  }
  else {
    printf("test failed.\n");
    return 1;
  }
}
```

----

3) Write a function named `print` which takes one argument of type `Point` and prints out the `x` and `y` values of that point. `print` should *not* be a member function of class `Point`, but a function that could have been written by a user of class `Point`.

Put the declaration of the `print` function in the file `print_pt.h`, and the definition in `print_pt.c`. Test your print function with the file `print_test.c` (which is also in the unit02/Labs directory.)

-------------------- FILE: Labs/print_test.c

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

main(int, char *[])
{
    Point p1, p2;

    p1.set_to(3, 5);
    p2.set_to(8, 2);

    printf("printing (3, 5): ");
    print (p1);
    printf("\nprinting (8, 2): ");
    print (p2);
    printf("\n");

    return 0;
}
```

----

4) In this exercise, you will make additions to the class `String` studied in the lecture. Return to the unit02 directory and modify the files there. Add member functions to class `String` so that users will be able to compare `Strings`. Add a function named `is_the_same_as` which returns a TRUE (non-zero) value if the invoking object is the same as the argument, and another named `is_different_from` which returns a TRUE (non-zero) value if the invoking object is different from the argument. If you feel ambitious, add `is_before` and `is_after` to tell if one string comes before or after the other in ASCII order.

Keep the code for your new member functions in the file `same.c`, so that you can copy them for later labs. Test your class with the 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("\nString::is_the_same_as ");
  if (h1.is_the_same_as(h2) && !h1.is_the_same_as(w))
    printf("works.\n");
  else
    printf("doesn't work.\n");

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

  return 0;
}

-----------------------------------------------------------
```

Exercise Answers

UNIT 2
Lab Exercises

In this unit, question 1 lets you compile a C++ program. Start with question 1 regardless of which class (**String** or **Point**) interests you more. Questions 2 and 3 let you start class **Point**, and question 4 asks you to enhance the **String** class discussed in the lecture.

1. Change to the *unit02/string* directory. The **String** class discussed in the lecture is in this directory. The class declaration is in the file *String.h*, and the implementation of the member functions is in the file *string.c*. Compile and run the program *use_string.c*, which is a sample use of class **String**.

SUMMARY	
DIRECTORY	unit02/string
DECLARATION	String.h
IMPLEMENTATION	string.c
TEST PROGRAM	use_string.c

FILE: **use_string.c**

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

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

   s.set_to("hello world");
   printf("Length of s is %d\n", s.length() );

   printf("Enter string t:  ");
   t.read();
   printf("Length of t is %d\n", t.length() );

   return 0;
}
```

2. Change to the *unit02/point* directory. Create a class **Point** that represents a pair of co-ordinates. Such a class might be used to identify positions on the terminal screen in a graphics program. Your class needs only three member functions: one (called **x**) to find the distance along the X axis, one (called **y**) to find the distance along the Y axis, and one, called **set_to**, to give values to the **Point**'s X and Y co-ordinates. (The function's first argument should be the X value, and the second argument the Y value.) You may assume that the X and Y values will be integers. You may use whatever private data members seem appropriate (such as two integers named **_x** and **_y**. Note that we should not give the data the same name as the member functions: adding underscores or capitalizing the names are two ways

to prevent this).

Put the class declaration in the file *Point.h*, and the source for the member functions in *point.c*, both in the *unit02/point* directory. Test your class with the test program *use_point.c*, which is located in the *unit02/point* directory. You can compile and execute this test program (**use_point**) by entering '**make prob2** or you can compile and execute it directly using the commands:

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

SUMMARY	
DIRECTORY	unit02/point
DECLARATION	Point.h (new)
IMPLEMENTATION	point.c (new)
TEST PROGRAM	use_point.c

FILE: **use_point.c**

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

main(int, char *[])
{
  Point p1, p2;

  p1.set_to(3, 5);
  p2.set_to(2, 7);

  if (p1.x() == 3 &&
      p1.y() == 5 &&
      p2.x() == 2 &&
      p2.y() == 7)
  {
    printf("test successful.\n");
    return 0;
  }
  else {
    printf("test failed.\n");
    return 1;
  }
}
```

3. Write a function named **print** which takes one argument of type **Point** and prints out the x and y values of that point. print should **NOT** be a member function of class **Point**, but a function that could have been written by a user of class **Point**.

   Put the declaration of the print function in the file *print_pt.h*, and the implementation in *print_pt.c*. Test your print function with the file *print_test.c* (which is also in the *unit02/point* directory). You can compile and execute this test program (**print_test**) by entering **'make prob3'** or you can compile and execute it directly using the commands:

```
$ CC -o print_test print_test.c point.c print_pt.c
$ print_test
```

SUMMARY	
DIRECTORY	unit02/point
DECLARATION	Point.h, print_pt.h (new)
IMPLEMENTATION	point.c, print_pt.c (new)
TEST PROGRAM	print_test.c

FILE: **print_test.c**

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

main(int, char *[])
{
     Point p1, p2;

     p1.set_to(3, 5);
     p2.set_to(8, 2);

     printf("printing (3, 5): ");
     print (p1);
     printf("\nprinting (8, 2): ");
     print (p2);
     printf("\n");

     return 0;
}
```

4. In this exercise, you will make additions to the class **String** studied in the lecture. Return to the *unit02/string* directory and modify the files as follows. Add member functions to the class **String** (declared in *String.h*) so that users will be able to compare **Strings**. Add a function named **is_the_same_as** which returns a TRUE (non-zero) value if the invoking object is the same as the argument, and another named **is_different_from** which returns a TRUE (non-zero) value if the invoking object is different from the argument. If you feel ambitious, add **is_before** and **is_after** to tell if one string comes before or after the other in ASCII order. If you add these two functions, change the corresponding lines from comments to statements in the file *str_same.c* that test these functions.

Put the implementation of your new member functions in the file *same.c* (the existing member functions are in the file *strings.c*). Test your newly modified class with the program *str_same.c*. You can compile and execute this test program (**str_same**) by entering '**make prob4**' 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	unit02/string
DECLARATION	String.h (modify)
IMPLEMENTATION	string.c, same.c (new)
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("\nString::is_the_same_as ");
  if (h1.is_the_same_as(h2) && !h1.is_the_same_as(w))
    printf("works.\n");
  else
    printf("doesn't work.\n");

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

// The following can be used to test the 'is_before' and 'is_after'
// member functions if you have included them in your String class.
//
//  printf("\nString::is_before ");
//  if (!w.is_before(h2) && h1.is_before(w))
//    printf("works.\n");
//  else
//    printf("doesn't work.\n");
//
//  printf("\nString::is_after ");
//  if (w.is_after(h2) && !h1.is_after(w))
//    printf("works.\n");
//  else
//    printf("doesn't work.\n");

  return 0;
}
```

UNIT 2

Lab Exercises (Answers)

1. To compile the program from question one, log in and run the commands:

```
cd unit02/string
CC -o use_string use_string.c string.c
```

2. The file *Point.h* contains the declaration for class **Point**:

FILE: **`Point.h`**

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

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

The file *point.c* contains the member functions:

FILE: **`point.c`**

```
#include "Point.h"

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


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

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

3. The declaration of the print function is in the (one line) header file *print_pt.h*:

FILE: **print_pt.h**

```
void print(Point);
```

The code for the function in the file *print_pt.c*:

FILE: **print_pt.c**

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

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

4. The functions **is_the_same_as** and **is_different_from** (as well as **is_before** and **is_after** if you chose to add them) must be added to the class:

FILE: **String.h**

```
const int max_string_length = 128;

class String {

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

     int is_the_same_as(String);
     int is_different_from(String);
     int is_before(String);
     int is_after(String);

     String substring(int start, int len);
     // return substring of invoking string,
     //  without changing original.

     String concat(String *);
     // return the concatenation of invoking string
     //  and the argument, without changing either.

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 code for the functions are in the file *same.c*:

FILE: **same.c**

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

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

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

int String::is_before(String s)
{
   return (strcmp(text, s.text) < 0);
}

int String::is_after(String s)
{
   return (strcmp(text, s.text) > 0);
}
```

NOTE: the solution contains the functions **is_before** and **is_after**. However, these two functions will **NOT** be carried forward to subsequent unit exercises.