Object-Oriented Programming in C++

Overloaded Functions

Objectives

Objectives

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

- Define the term "overloaded"
- Explain the importance of overloading
- Overload a function name
- Provide default values for function arguments
- Call a C function from C++

Overloading

If a function name or operator has different meanings when applied to different types, that name or operator is *overloaded*.

Function names are overloaded by declaring several functions with the same name but different argument types. Note that the functions must differ in the number or type of arguments, or in the type of invoking object, so that C++ can identify which function should be called.

Overloading relieves the programmer from the burden of making up different names for several functions that all take the same action on arguments of different types. It also lets you choose the most meaningful name for a function, without fear that the name will be confused with a function from an unrelated library.

Macros that rely on overloaded function names or operators may be used with any type for which that function or operator is defined. For example, the `average` macro on the facing page can be used with arguments of type `int` type `float`, or any other type for which addition, and division by an integer have been defined. Code that works for many different types of arguments is know as *polymorphic* code. We will see how to create polymorphic functions later in this course.

Overloading

C++ allows overloading of function names and operator symbols:

- One name or symbol for several functions
- Functions must have different parameter types or be members of different classes
- Function selection based on types of arguments and invoking object

Overloading

- allows more meaningful function names
- allows *polymorphic* code

```
#define max(a, b)      ( (a) > (b) ? (a) : (b) )
#define min(a, b)      ( (a) > (b) ? (b) : (a) )

#define average(a, b)  (( (a) + (b) ) / 2)
```

Overloading a Function Name

We may need to concatenate either a String variable or a quoted group of characters onto a String, so we will provide a second concatenation function. In C, we would have to make up 2 different names for these 2 functions, and we would have to remember to use the right name in each function call. In C++, we can simply write 2 functions named concat. The C++ compiler will select the appropriate function by looking at the type of the argument.

Overloading a Function Name

```
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);

  String concat(String *);  // for s1.concat(&s2);
  String concat(char *);    // for s2.concat("text");

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

String::concat

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

String String::concat(String *other)
{
    // original concat function
}

String String::concat(char *ptr)
{
    String both;

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

    return both;
}
```

Using String::concat

When we use the function name `concat`, C++ will select one of the `concat` functions based on the types of the invoking object and arguments.

Using String::concat

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

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

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

    name = firstname.concat(" ");
    name = name.concat(&lastname);
    // or: name = firstname.concat(" ").concat(&lastname);

    printf("name is:  ");
    name.print();

    return 0;
}
```

Overloading a Non-member Function

With releases 1.0, 1.1, and 1.2 of C++, the keyword "overload" is used to introduce an overloaded function name, if that function is neither a member function nor an operator function (we will study operator functions later). To overload the name sentence on one of those releases, we must place the declaration

```
overload sentence;
```

before the declaration of any of the sentence functions.

Overloading a Non-member Function

```
#include "String.h"

String sentence(String words, char *punctuation);
String sentence(String words);

main(int, char *[])
{
    String statement, question;
    statement.set_to("Hello, Zaphod");
    question.set_to("Do you have any tea");

    sentence(statement).print();
    sentence(question, "?").print();
    return 0;
}

String sentence(String words, char *punctuation)
{
    String result;
    result = words.concat(punctuation);
    return result;
}

String sentence(String words)
{
    String result;
    result = words.concat(".");
    return result;
    // or, return sentence(words, ".");
}
```

Default Function Arguments

When one function can easily be written in terms of another, as in the case of the `sentence` functions on the previous page, it may be possible to use a default argument to one of the functions instead. You can provide default values for function arguments by giving an initial value for the formal parameter in the function declaration, as shown on the facing page. If actual arguments are not provided in the call, as in the case of `sentence(statement)`, C++ will use the default value.

Defaults may be provided for several arguments, but they may only be used "from the right" of the argument list. That is, there is no way to use a default for one argument without using the defaults for all arguments to its right:

```
void example(int arg1 = 1, int arg2 = 2, int arg3 = 3);

// legal calls:
example(4, 5, 6); // no defaults
example(4, 5);    // default for arg3
example(4);       // defaults for arg2 and arg3
example();        // default for all arguments

// example(4, ,6) IS ILLEGAL
```

Default Function Arguments

```
#include "String.h"
```

*String sentence(String words, char *punctuation = ".");*

```
main(int, char *[])
{
    String statement, question;
    statement.set_to("Hello, Zaphod");
    question.set_to("Do you have any tea");

    sentence(statement).print();
    sentence(question, "?").print();

    return 0;
}

String sentence(String words, char *punctuation)
{
    String result;
    result = words.concat(punctuation);

    return result;
}
```

Calling C Functions

Since C++ must be compatible with existing program linkers, it can not produce ".o" files in which one name is used for several functions. To ensure that each function has a unique name, the C++ compiler extends all function names in the ".o" files it produces.

The C compiler, however, does not need to extend function names. If C++ is to call a function that was compiled with a C compiler, the C++ compiler must generate a call without an extended function name. It will do this for all functions declared as **extern** "C".

Note that the C++ header files for the standard C libraries use the **extern** "C" declaration.

Calling C Functions

```
extern "C" double sqrt(double);

extern "C" {
     int printf(const char * ...);
     int puts(const char *);
}

main(int, char *[])
{
     float f = sqrt(2);
     puts("hello, world\n");
     printf("sqrt(2) = %f\n", f);

     return 0;
}
```

Summary

In C++, you can create many functions with the same name (or operator symbol), as long as they have either: (a) different types of invoking objects, (b) different types of arguments, or (c) different numbers of arguments, so that C++ can tell them apart when a call is made.

Summary

C++ will select a function based on:

- The name of the function
- The number of arguments
- The type of the invoking object
- The types of the arguments

You can provide default arguments.

Lab Exercises

UNIT 3

Lab Exercises

1. This exercise demonstrates why overloading makes it easier to use two or more libraries together in one program. Change to the *unit03/point* directory. The files *print_pt.c* and *print_pt.h* contain the function you developed as part of the Unit 2 Lab Exercises. This can be considered as the first library. The files *print.c* and *print.h* are new and contain additional functions named **print**, to print out objects of type int, float, or char *. This is the second library.

   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_test print_test.c point.c print_pt.c \
       print.c
$print_test
```

   Note that the test program uses the **print** functions from both libraries (the one declared in *print.h* and the one declared in the new *print_pt.h*). Does this cause any problem? Would it cause a problem in a C program?

SUMMARY	
DIRECTORY	unit03/point
DECLARATION	Point.h, print_pt.h (library 1), print.h (library 2)
IMPLEMENTATION	point.c, print_pt.c (library 1), print.c (library 2)
TEST PROGRAM	print_test.c

FILE: **print.h**

```
// if using C++ 1.2 or before,
// then use "overload print;" here

void print (char *);
void print (double);
void print (int);
```

FILE: **print_test.c**

```
#include "Point.h"
#include "print_pt.h"
#include "print.h"

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

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

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

    return 0;
}
```

---

2. Change to the *unit03/string* directory. The file *same.c* contains the **String** member functions from the Unit 2 Lab Exercises. The file *string.c* has been updated to include the second **concat** function discussed in the lecture. The declarations for these functions have also been added to the *String.h* header file. Add two more member functions to the *same.c* implementation file and the *String.h* header file to allow the comparison of Strings and character arrays. The new member functions should be named **is_the_same_as** and **is_different_from**, and each should have one parameter of type **char *** . Test them with the new *str_same.c* program in the *unit03/string* directory.

   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 str_same str_same.c string.c same.c
$ str_same
```

SUMMARY	
DIRECTORY	unit03/string
DECLARATION	String.h (modify)
IMPLEMENTATION	string.c, same.c (modify)
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");

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

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

  return 0;
}
```

UNIT 3

Lab Exercises (Answers)

1. The **print** functions from *print.c* and the **print** function in *print_pt.c* can be used together in the same C++ program. This would not be possible in C.

2. The new **String** class:

FILE: **String.h**

```
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);

     String concat(String *);  // for s1.concat(&s2);
     String concat(char *);    // for s2.concat("text");

     int is_the_same_as(String s);
     int is_different_from(String s);

     int is_the_same_as(char *);
     int is_different_from(char *);

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 new **String** member functions:

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_the_same_as(char *s)
{
   return !strcmp(text, s);
}

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