Unit 1

Object-Oriented Programming in C++

Introduction

Objectives

Objectives

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

- Distinguish object-oriented from procedure-oriented programming.
- Define the terms data abstraction and data encapsulation, and explain how they relate to C and C++.
- List some of the advantages of object-oriented programming.

What is C++?

C++ is an extension of the C language that is designed to support a kind of programming known as "object-oriented programming." An understanding of the basic principles of object-oriented programming is essential for anyone who wants to obtain the full benefit of C++.

What is C++?

C++ is an extension of the C language that:

- Provides facilities to support object-oriented programming
- Compiles C code with little or no change
- Maintains the overall character of C:
  - — New features maintain C's high efficiency
  - — New features allow "low-level" programming
  - — New features let the programmer "bend the rules"

Procedure-Oriented Programming

We will introduce object-oriented programming (often abbreviated O.O.P.) by contrasting it with procedure-oriented programming (P.O.P.). O.O.P. and P.O.P. are two ways of thinking about and organizing programs: both help you divide a program into pieces that can be written independently and combined later. They differ in the kinds of pieces you create.

Each piece must be written by one or more programmers, whom we call the *authors*. Other programmers (the *users)* use the pieces to construct other pieces or the complete program itself. A programmer may use existing pieces of code while acting as the author of a new piece. For example, if you use the C standard I/O library and the math library to create a calculator program, you are the author of the calculator program and the user of both libraries.

Individual pieces of code are most useful when they let the users think of the piece in *abstract* terms. That is, the users can think in high-level terms when they use the code, without having to worry about low-level details of how the piece is implemented. An abstraction separates what the user needs to know to use a piece of code from what the author had to know to write the code. For example, when you use the `sin` function, you can think in terms of the mathematical idea of the sine of an angle, without worrying about how the library function computes the value.

Note that, for the purposes of this discussion, we will not distinguish between procedures and functions.

Procedure-Oriented Programming

Procedure-oriented programming

- a way of thinking about and organizing programs
- programs are made up of *procedures:*
  - — written by one or more *authors*
  - — used by many *users*
  - — known as *procedural abstractions* if use does not require knowledge of implementation

Language Support for Procedure-Oriented Programming

Traditional languages have many features to help you create procedural abstractions.

- procedure names
- restricted scope of local variables
- argument & return value copying

It is possible to use many of the techniques of procedure-oriented programming with a language that lacks the above features, but many of the advantages will be lost, since the code can not fully reflect the design principles.

Language Support for Procedure-Oriented Programming

Language features of traditional languages (like C) help you create procedural abstractions:

- Function names
- Limited scope of local variables
- Automatic function argument and return value copying

Object-Oriented Programming

Programs contain both data and code to manipulate those data. Procedure-oriented programming focuses on defining abstractions that correspond to the actions taken by the code. Object-oriented programming focuses on defining abstractions that represent the data being manipulated.

Procedural abstractions allow many programmers to use a procedure without knowledge of the details known to the procedure's authors. Data abstractions allow many programmers to use a data type without knowledge of the details known to the data type's authors. For example, you can use the data type `float` in a C program without worrying about how floating point numbers are represented on the machine that will run your program.

When several classes have similar features, they can be defined in a simple, flexible, and efficient manner using *inheritance*.

Object-Oriented Programming

Object-oriented programming (O.O.P.)

- another way of thinking about and organizing programs
- programs made up of interacting pieces of data (*objects*)
- groups of objects defined by *classes* (types)
  - — classes written by one or more *authors*
  - — classes used by many *users*
  - — known as *data abstractions* if use does not require knowledge of implementation
- groups of classes defined with *inheritance*

Language Support for Object-Oriented Programming

Object-oriented languages have features that help you create data abstractions and describe the relationships between these data abstractions.

Language Support for Object-Oriented Programming

Features of *object-oriented languages* (like C++) let you:

- create data abstractions (classes)
- use user-defined types like built-in types
- create groups of related types using inheritance

Object-oriented languages usually contain features that allow the creation of procedural abstractions.

Programming with Abstractions

There are many reasons to build a program with abstractions.

If you build a program out of abstract pieces, the implementation details of each piece are hidden in that piece. This makes the program easier to write, since you will not be distracted by the implementation details of other parts of the program. For example, you do not have to stop and think about the implementation of the `sin` function when you call it. This also makes programs easier to read, since each part of the program is concerned with only one task.

Programs built from abstractions are also easier to maintain, since changes to one piece of code will not "break" other parts of the program.

If you can understand a part of a program in abstract terms, without extensive study of its implementation, you can easily re-use that piece of code in a later program.

One of the most significant advantages gained by building a program from abstractions is that each piece of the program corresponds to one of the programmer's ideas. It is difficult to describe exactly how it improves a program, but programmers who are familiar with this correspondence often point to it as one of the most important aspects of "good" code.

Programming with Abstractions

Building a program from abstractions:

- localizes implementation details
- eases program maintenance
- allows easy re-use of parts of a program
- aligns pieces of a program with programmer's ideas.

Advantages of Object-Oriented Programming

Object-oriented programming lets you take full advantage of programming with abstractions. Procedure-oriented programming allows only one kind of abstraction, so many of the benefits listed on the previous page can only be partially achieved with procedure-oriented programming alone.

Object-oriented programming also lets you create groups of similar types by using inheritance. When you do this, you can write code that will work for any of the types in the group. This ability can greatly simplify programs, since individual functions can work for many types of arguments.

You can easily extend an object-oriented program by adding new types into an existing group of similar types. When you do this, code that you wrote to work with the original group will also work with the new type.

Advantages of Object-Oriented Programming

Advantages of object-oriented programming:

- all advantages of abstraction apply to data
- groups of related data types can be created easily with inheritance
- new types can easily be added to a group

Data Abstraction

A data abstraction is a type defined in terms of a list of operations (which we will call the *defining operations*).

The built-in types of the C language (`float`, `int`, etc.) are data abstractions. C defines many operations for `floats`, but leaves the implementation of floating point calculation to the compiler writer. You can write a C program without any knowledge of how floating point numbers are represented on the machine that will run your program.

C++ lets you define new types as data abstractions.

Data Abstraction

A *data abstraction*

- is a type defined in terms of operations
- for example, the C type `float`
  - — addition
  - — subtraction
  - — multiplication
  - — division
  - — comparison
  - — assignment

C++ allows the definition of new data abstractions.

Data Encapsulation

If a language allows only the defining operations of a type to work with the implementation of objects of that type, the language is said to enforce *data encapsulation.* Any other operations must be written in terms of the defining operations (we are using the term operations to mean both the traditional operators like +, -, etc., and user-defined functions, like `average` on the facing page).

For example, if we want to work with floating point numbers, we must use the operations; we can not access the bits that represent the exponent or mantissa directly.

Data Encapsulation

Data encapsulation

- only defining operations can access implementation
- other operations must use defining operations:

```
float average(float a, float b)
{
    return (a + b)/2;
}
```

Benefits of Data Encapsulation

C enforces data encapsulation for built-in types (but it lets you get around the rule by type-casting pointers).

In C++, you can define a class with encapsulated data (in which case, you can still get around the rule by type-casting pointers).

Data encapsulation ensures that only a small part of any program depends on the representation of any one type. If the representation changes, only the defining operations for the type need to be modified.

Benefits of Data Encapsulation

C enforces data encapsulation for built-in types.

C++ allows user-defined types with data encapsulation

Benefits of data encapsulation:

- only defining operations depend on representation.
- change to representation only affects defining operations

String Manipulation in C

We have seen that the built-in types of the C language are abstract, and that C provides data encapsulation for them. The C language does not provide a built-in type for string manipulation; null-terminated arrays of characters are used for that purpose. Programmers writing programs often need to remember that this is the way strings are represented in C: you often have to remember to allocate enough space for the null byte, or store a null at the end of an array.

String Manipulation in C

In C:

- strings are represented with null-terminated arrays of `char`
- programmers must be aware of representation
- code often has to work with implementation details

String Manipulation in C
Example

Much of the code on the facing page depends on the representation of strings in C. If we decided that we would rather represent strings with a count of characters followed by the characters themselves, how much of the program on the facing page would we need to change? What fraction of existing C programs would need to be changed?

String Manipulation in C
Example

```
/* concatenate two strings -- C version.  */
#include <string.h>
#include <stdlib.h>

char *name(char *first, char *last)
{
     unsigned tot_length;
     char *nam;
     tot_length = strlen(first) + strlen(last) + 2;
     nam = malloc(tot_length);
     if (nam == NULL) {
       fprintf(stderr, "Out of storage\n");
       exit(1);
     }
     strcpy(nam, first);
     strcat(nam, " ");
     strcat(nam, last);
     return nam;
}

main(int argc, char *argv[])
{
     char *full_name;
     full_name = name("Zaphod", "Beeblebrox");

     /* work with full_name */
     free(full_name);
     return 0;
}
```

String Manipulation in C++

We will see that it is possible to create a data abstraction to represent strings in C++. Users of this abstraction will not need to know its implementation details, and they will not write code that depends on that implementation.

String Manipulation in C++

Later, we will create a C++ class `String` to represent strings:

- we must choose an implementation
- users can ignore our implementation
- code that depends on our implementation
  - — is isolated in the defining operations of our class
  - — does not creep into code that works with `Strings`.

String Manipulation in C++
An Example

The code on the facing page does not depend on any particular representation of strings. Class `String` is declared in the header file `String.h`. The header file and the associated library contain all the implementation details of class `String`. If we choose to change the implementation, we will not need to change the code on the facing page (although we will have to re-compile it).

String Manipulation in C++
An Example

```
/*  concatenate two strings -- C++ version.  */

#include <String.h>

String name(String first, String last)
{
    return first + " " + last;
}

main(int, char *[])
{
    String full_name;
    full_name = name("Zaphod", "Beeblebrox");

    // work with full_name

    return 0;
}
```

Summary

Summary

Abstractions

- allow use without knowledge of implementation
- make code more modular

Object-oriented programming

- allows the creation of data abstractions
- allow new types to be used like built-ins
- allow creation of groups of related types with inheritance

Exercises 1 Ex

Object-Oriented Programming in C++

Lab Exercises

Lab Exercises

UNIT 1

Lab Exercises

1. The FILE structure in C's standard I/O library is one example of data abstraction in C. What functions can you think of that work with FILEs? Do you need to know anything about the FILE structure itself to use these functions? In what ways do FILEs differ from other objects in C?

2. C's standard library contains many string processing functions. Are strings an abstract type in C? Why or why not?

UNIT 1

Lab Exercises (Answers)

1. The FILE structure can be manipulated with the functions **fprintf**, **fputc**, **fputs**, **fscanf**, **fgetc**, **fgets**, **ftell**, **fseek**, and other functions from the stdio library, without knowledge of the FILE structure (although one should understand the concept of buffered I/O). FILEs are different from other types of objects because one should not create a local variable of type FILE.

2. The string manipulation functions from C's standard library do not allow the use of strings as an abstract type, because programmers must be aware of the representation of strings (as NULL terminated arrays of char) to use work with strings.

INTRODUCTION TO LAB EXERCISES

Under your login directory you will find the *unit02* through *unit18* directories and the *pre_windows* directory. The *pre_windows* directory will not be used until Unit 12. Each *unit***nn** directory contains subdirectories (e.g., *string*) for each type of class (e.g., **String**) used in that unit's lab exercises. In addition, each *unit***nn** directory contains subdirectories called *lecture* and *solutions*. The *solutions* subdirectory contains the updated files (i.e., "answers") for this unit. The *lecture* subdirectory contains the files used as examples during the lecture along with a *Makefile* and a file named *RUN*. If the system you are using supports the **make** command, you can go to the *lecture* directory and enter '**make**' to compile and execute the test programs discussed during the lecture. If **make** is not supported, the *RUN* file can be executed to compile the test programs. You are strongly encouraged to use the **make** command because it is far more efficient than the **RUN** script. The *lecture* directory and its contents are provided in case you want to experiment with the examples used in class.

Each class directory (e.g., *string*) will also contain a *Makefile* for your convenience in compiling the Lab Exercises associated with that class. The instructions for each exercise will tell you the appropriate **make** command (with any necessary arguments). When you have completed all of the exercises in any class directory or unit directory, you can execute the '**make clean**' command to remove all of the object files from that directory and all sub-directories. This will keep the amount of disk space used to a minimum.

In many units, you may 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 of the early 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 the answers to the **Point** questions in this unit, and future lab exercises for class **String** will depend on the answers to the **String** labs in this unit. Therefore, you may find it helpful to pick one of these two classes, and, in each unit, do the labs for that class first.

Each exercise will contain a SUMMARY table specifying the name of the file(s) where the class is declared, the name(s) of the file(s) where the class member functions are implemented, and the name of the file(s) containing the test program(s). A test program represents the application program normally written by the user. The SUMMARY table will also indicate the status of each file: '(modify)' means that the file exists but you must modify it in some way to complete the exercise, '(new)' means that you must write or provide the entire contents of the file. If neither is specified, you must use the file but do not have to make any changes to it. All files specified as '(new)' in the SUMMARY table exist in the appropriate directory but contain only comments, i.e., no code. Thus, you do not create any '(new)' file; you only add the appropriate contents (code, declarations, etc.).

If the C++ compiler is not invoked with the "CC" command, you must specify the "**CC=***compiler_name*" argument on the **make** command line, i.e.:

make CC=*compiler_name* **prob1 #instead of 'make prob1'**

Each exercise specifies the sequence of commands to compile and execute the test program. Executing the test program by entering its name assumes that the current directory is specified as part of the **PATH** shell variable. If this is not the case, you must either add the current directory (".") to the **PATH** variable or execute the test program by entering "**./prog_name**" instead of just "**prog_name**".