C++ Programming

Student Workbook

itcourseware
THE SOURCE FOR IT PROFESSIONALS℠

C++ Programming

Published by ITCourseware, LLC, 7245 South Havana Street, Suite 100, Centennial, CO 80112

Contributing Authors: Andrew Boardman, Todd Gibson, Evans Nash

Editor: Danielle Waleri

Assistant Editor: Ginny Jaranowski

Special thanks to: Many instructors whose ideas and careful review have contributed to the quality of this workbook, including Brandon Caldwell, Rick Sussenbach, and Lynwood Wilson, and to the many students who have offered comments, suggestions, criticisms, and insights.

Contents

Chapter 2 - Classes

Notes

User Requirements for the Application

1. Quality Assurance personnel need to be able to record bugs they find when testing products under development. Information such as when the bug was found, the severity of the bug, and a description of the bug, need to be captured.

2. As developers fix bugs, they'll need to enter their fixes into the system. The information captured for the fix includes the developer that fixed the bug, which bug was fixed, and the date of the fix.

3. The QA organization currently has an inefficient manual system of tracking memos and status reports. They would like to automate this by entering memos and status reports on-line and having them automatically routed and tracked. In addition to the text, these memos and reports also have a date and time of creation.

4. There needs to be support for administrative functions such as these: printing out the list of bugs, fixes, and status reports; archiving old information; and searching for specific reports and fixes.

Your role will be to create the tools that someone else will use to build this application.

Scenario

Bug reports and bug fixes will need to have a date associated with them. Our first step is to develop a `Date` type.

We need to implement our `Date` type so that it can be used with operations that we provide. Furthermore, any code using our `Date` type should not be allowed to directly access its internal details.

What data should our `Date` type have?

Chapter Objectives

- Define a class with data and methods.
- Define what an object is and be able to declare one.
- Describe what a data member is.
- Describe what a method is and how to invoke it.
- Describe what a namespace is.
- Use access specifiers to enforce encapsulation.

Notes

Let's create the `Date` structure and a `main()` to test it out:

Date.cpp

```
struct Date {
     int day;
     int month;
     int year;
};

void main() {
     Date today;
     today.day = 2;
     today.year = 1998;
     today.month = 10;
     today.day = today.day + 1;
}
```

Remember, you need the semicolon after a `struct` definition because you could make a variable of that type, for instance:

```
struct Date {
     int day;
     int month;
     int year;
} globaldate;
```

Creating a Data Structure

- A `struct`'s data members describe the individual features of a complex type of data.

```
struct Date {
     int day;
     int month;
     int year;
};
```

  - Members of the structure can be built-in types or other structures.

- Once you've defined a `struct`, you can declare variables representing distinct instances of the `struct`'s type.

```
Date today, tomorrow;
```

- Use dot notation to access members of the structure.

```
today.day = 2;
today.year = 1998;
today.month = 10;
today.day = today.day + 1;
```

- What functions are we going to need to service this data structure?

Notes

Let's add in a `set()` method with its code:

Date.cpp

```
struct Date {
     int day;
     int month;
     int year;
     void set(int m, int d, int y);
};

void Date::set(int m, int d, int y)
{
     day = d;
     month = m;
     year = y;
}

void main()
{
     Date today;
     today.set(10,2,1998);
     today.day++;
}
```

Methods

- `struct`s have always been part of the C language.
- C++ adds the ability to define functions within a `struct`.
- To call these functions, use dot notation, just like structure data.

  ```
  Date today;
  today.set(10, 2, 1998);
  ```

- A function defined together with its data is a *method*.
- Methods are also called *member functions*, *operations*, or *messages*, depending on a programmer's background.
  - "Member function" is often used by people with a C/C++ background.
  - "Method" or "message" is more common from people with a pure OO background.
- When writing the implementation of the method, you must specify which structure the method belongs to.
  - The name of a method is preceded by the name of the structure, plus the *scope operator* (`::`).

    ```
    void Date::set(int m, int d, int y)
    ```

    - This is the `set()` method for `Date` (other structures might also have `set()` methods).

Notes

Both of these `main()` functions modify `today`'s `day`, `month`, and `year`.

```
void main()
{
     Date today;

     today.day = 2;
     today.year = 1998;
     today.month = 10;
     today.day = today.day + 1;
}
```

```
void main()
{
     Date today;

     today.set(10, 2, 1998);
     today.day++;
}
```

Object Scope

- Methods always run in the context of an invoking object.

```
Date today;
today.set(10, 2, 1998);
```

  - ☐ `set()` is called within the context of the `Date` object `today`.
  - ☐ Within `Date`'s `set()` method, `day = d` sets `today`'s `day` member to `2`.

```
void Date::set(int m, int d, int y)
{
     day = d;
     month = m;
     year = y;
}
```

- Within any `Date` member function, there will always be available a `day`, `month`, and `year` belonging to the invoking object.

```
Date tomorrow;
tomorrow.set (10,3,1998);
```

  - ☐ In this case, the `set()` method sets `tomorrow`'s `day`, `month`, and `year`.

Notes

Now, let's add a `display()` method with its code. We must include `iostream` and the information about the `std` namespace:

Date.cpp

```
#include  <iostream>
using  namespace  std;

struct  Date  {
     int  day;
     int  month;
     int  year;
     void  set(int  m,  int  d,  int  y);
     void  display(void);
};

void  Date::set(int  m,  int  d,  int  y)
{
     day = d;
     month = m;
     year = y;
}

void  Date::display(void)
{
     cout  <<  month  <<  '/';
     cout  <<  day  <<  '/';
     cout  <<  year  <<  endl;
}

void  main()
{
     Date  today;

     today.set(10,  3,  1998);
     today.display();
     today.day++;
     today.display();
}
```

C++ Input and Output

- C++ I/O is done through *streams*, which provide you with a consistent interface to consoles, files, etc.
  - `cout` is an output stream connected to the program's standard output destination.
  - `cin` is an input stream connected to the program's standard input source.
  - `cerr` is an output stream connected to the program's standard error destination.
- Use the `<<` operator to send data to an output stream.

```
cout << "Hello, world!\n";
cerr << "Can't find my file!" << endl;
```

  - `endl` sends a newline to the output stream, and then flushes the stream.
- Use the `>>` operator to get data from an input stream.

```
int i;
cin >> i;
```

- At least for testing purposes, most of your classes, or structures, will need an easy way to be output, such as a `display()` method.

Notes

The Evolution of Header Files

Older style C and C++ header files have a `.h` extension. These are NOT the same as the new header files without the extensions. The main difference is the use of namespaces in the new C++ header files.

Older compilers do not understand the use of namespaces. So newer compilers are distributed with two sets of header files.

You should NOT mix the usage of old and new header files.

Examples:

C++ header files no longer have the `.h` extension.

Old C++ header files	New C++ header files
`iostream.h`	`iostream`
`fstream.h`	`fstream`
`limits.h`	`limits`

C header files no longer have the `.h` extension, and start with the letter `c`.

Old C header files	New C header files
`stdio.h`	`cstdio`
`stdlib.h`	`cstdlib`
`string.h`	`cstring`

Namespaces

- All of the classes that are part of the standard C++ library are in a *namespace* called `std`.
  - ◻ Namespaces are one of the features that C++ uses for modularity.
- Including a header file makes all the constructs in the file available.
  - ◻ A statement in the format of `using namespace` followed by the namespace name causes all of the parts of that namespace to be integrated into the global namespace.

    ```
    using namespace std;
    ```

    - It may be desirable to only include those that are going to be used.
  - ◻ In this course, we will provide a general `using` statement for simplicity's sake.
  - ◻ To use `cout`, `cin`, and `endl` without the `std::` notation, do the following:

    ```
    #include <iostream>
    #include <fstream>
    using std::cout;
    using std::cin;
    using std::endl;
    ```

  - ◻ Without the `using` line, the compiler wouldn't recognize standard classes such as `ostream`, the data type for `cout` and `cerr`.

Notes

The built-in types that you use, such as `float`, are abstract data types. Instead of having methods to modify their data, you have operators such as `=`,`+`, and `–`.

```
float  f;
f  =  3.14;
f  =  f  +  6.8;
```

A `float` is abstract because you, as a user of a `float`, don't care that it is stored in IEEE format.

Data Abstraction

- When you separate the way you access and modify data from the way the data is stored, you're using *data abstraction*.
  - ☐ This usually means that the user of a data structure cannot (or should not) directly access its data.
  - ☐ Methods are provided to get and set the data values.
- This separates *user* code from *implementation* code.
  - ☐ User code is any code that is not part of the data structure.
  - ☐ Implementation code is any method defined within the structure.
- If only the implementation code has access to the data, we have achieved data abstraction.
- Data abstraction makes maintenance easier because the data and the code that can modify it are grouped together.

Notes

We often speak of *user code*. This is any code that is **not** part of your structure. For example, the `main()` function you wrote uses your `Date` structure.

The keywords `public` and `private` can appear in any order and as many times as desired.

For example:

```
struct Date {
public:
  void set(int m, int d, int y);
private:
  int day;
  int month;
public:
   void display(void);
private:
  int year;
};
```

Investigate:

Given that C++ is a superset of C, can you think of any ways of getting around `private`? (Keeping in mind that it's a bad idea!)

Enforcing Data Encapsulation

- You encapsulate data within a structure by restricting which methods have access to it.
- C++ provides *access specifiers* to identify which methods in an application can access the various parts of a structure.
- The `public` and `private` access specifiers enforce data encapsulation.

```
struct Date {
private:
  int day;
  int month;
  int year;
public:
  void set(int m, int d, int y);
  void display(void);
};
```

- Members of a structure declared to be `public` can be used from *any* code in an application.
- Members that are declared `private` can only be used by methods within this structure (called member functions).
    - ☐ So `public` member functions are used to access `private` data!

Notes

Date.h

```
#ifndef _DATE_H_
#define _DATE_H_ 1

struct Date {
public:
      void set(int m, int d, int y);
      void display(void);
private:
      int day;
      int month;
      int year;
};

#endif // #ifndef _DATE_H_
```

Date.cpp

```
#include <iostream>
using namespace std;

#include "Date.h"

void Date::set(int m, int d, int y)
{
      day = d;
      month = m;
      year = y;
}

void Date::display(void)
{
      cout << month << '/';
      cout << day << '/';
      cout << year << endl;
}
```

main.cpp

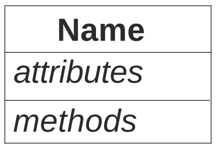
```
#include "Date.h"

void main()
{
      Date today;

      today.set(10, 3, 1998);
      today.display();
      today.set(10, 6, 1998);
      today.display();
}
```

File Organization

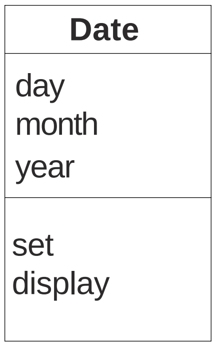
- Files are divided up for organization and compile efficiency.
- Structures like `Date` are often put into two files:
  - The *header* which has the declaration.
  - The *source file* which has the code for the methods (the definition).
- `Date.h` — the structure declaration.
- `Date.cpp` — the structure definition (implementation code).
- `main.cpp` — the application (user code).
- Divide `Date.cpp` into `Date.h`, `Date.cpp`, and `main.cpp`.
  - Don't forget! `#include` "`Date.h`" in `Date.cpp` and `main.cpp`.

Notes

Notation for classes:



For example:



Either one, or both, of the bottom two compartments are optional when drawing a Class Diagram.

Classes in C++

- *Class* is the OO term for a data type that uses data encapsulation and data abstraction.
- The C++ keyword `class`, like `struct`, declares the data and methods of a class.
  - Semantically, the only difference between a `struct` and a `class` is that a `struct`'s members are `public` by default, and a `class`'s members are `private` by default.
- Use a `class` when creating a data type that encapsulates an interface and its implementation.
- Use `struct` when a structure in the "traditional C" sense is needed — typically only for data.
- Let's turn our `Date` into a `class`.

  From:

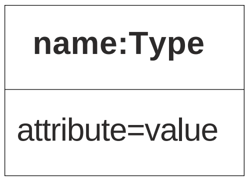
  ```
  struct Date {
  ...
  } ;
  ```

  To:

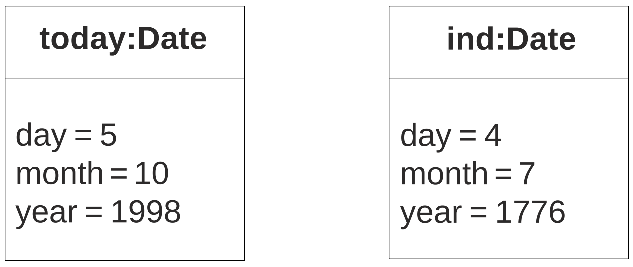
  ```
  class Date {
  ...
  } ;
  ```

Notes

Notation for objects:



For instance:



The second compartment is optional when drawing Object Diagrams.

Objects

- An *object* is an instance of a class; so, when you make a variable of type `Date`, you're making a `Date` object.

```
Date today;
today.set(10, 2, 1998);
```

  - In this case, `today` is an object of type `Date`; a `Date` object.

- Each `Date` object has its own `day`, `month`, and `year` values.

```
Date ind;
ind.set (7,4,1776);
```

  - `Date`s in different memory locations are different `Date` objects.

Notes

this Pointer

- For normal methods, the `this` pointer is a pointer to the invoking object.

- When a method is called, an additional parameter, the `this` pointer, is passed.

    - The call works like a standard C function, but the `this` pointer is hidden and never declared in the method header.

    - The `this` pointer gives access to the object's data.

        - A method call such as `christmas.set(12,25,2001)` would cause the header of the effective call to look like this (with `this` pointing to `christmas`):

```
void set( Date* this, int month, int day, int year)
```

    - When a normal function is called, nothing special is added.

Notes

Labs

1. Add a new method called `increment()` to your `Date` class that adds one day to its `Date` object. Code for the simplest case right now (ignore leap years, ends of months, etc.).

2. Add a `decrement()` method to your `Date` class too. Again, code for the simplest case.

3. Write code in your `main()` function to test your new methods. Can you use them together to easily test each other?

4. Create a `Time` class that keeps track of the `hour` and the `minute`. Write a `set()` method that takes as its arguments an hour and a minute.

5. Add a `display()` method to your `Time` class.

6. Look up `setw()`, `cout.width()`, and `cout.fill()` in the Reference Sheets appendix. Use those functionalities of C++ I/O to make sure your `Time` class displays `12:05` correctly.

7. (Optional) A year is a leap year when it is divisible by 4 but not divisible by 100 (except when it is divisible by 400). There are 31 days in January, March, May, July, August, October, and December.

   Give your `Date` class data sanity checking, and make sure your `increment()` and `decrement()` methods work correctly.

8. (Optional) Add data sanity checking to your `Time` class now. How are you going to differentiate between a.m./p.m.?

9. (Optional) Leap year doesn't apply to dates before September 1752 (that's when it started). Make sure your `Date` class doesn't try to apply leap year rules before then. Also, 9/3/1752 through 9/13/1752 don't exist; make sure your `Date` class won't allow them to be used.

Notes

Chapter 11 - Overloaded Functions

Notes

Scenario

Currently, once `Report`, `Bugreport`, and `ExternalBugreport` objects are created, there is no good way to set their data to different values. We would like to be able to do this with a `set()` method like we have for `Time` and `Date`. A `Report` `set()` method would probably need to set `rep_date` and `rep_time` with new `Date`s and `Time`s. So we'll need a `set()` method in `Date` that takes another `Date`, and a `set()` method in `Time` that takes another `Time`.

Chapter Objectives

- Describe what an overloaded function is.
- Know what conditions are required for overloading.
- Describe how `const` affects overloading.
- Declare and use default arguments.

Notes

Function Overloading

- Function overloading means two functions have the same name but take different arguments.
  - In this case, let's create a `set()` method for the `Date` class that takes a constant reference to a `Date` object.

    Date.h

    ```
    ...
    class Date {
    ...
        void set(int m, int d, int y);
        void set(const Date &dt);
    ...
    };
    ...
    ```

  - The code will simply copy the values from the `Date` passed.

    Date.cpp

    ```
    ...
    void Date::set(const Date &dt)
    {
        day = dt.day;
        month = dt.month;
        year = dt.year;
    }
    ```

Notes

Using Overloaded Functions

- Using an overloaded function is just like using any other function.
  - The only difference is the number and types of arguments you pass in.
  - The compiler takes the arguments you have given to it, and tries to match it to a function prototype.
    - If it doesn't find a function that matches exactly, the compiler will try implicit casting with the available methods.
  - This means that you can't have two overloaded functions with the same name that take the same arguments.
    - The number and type of the arguments is the only way the compiler knows they are different.
- We can use `Date`'s overloaded `set()` method to copy a `Date` object to an existing date object.
  - We couldn't do this with the copy constructor, because constructors create an object, and we have already created one.

```
void main()
{
...
    Date ind(7, 4, 1776), ind_copy;
    ind.display();
    ind_copy.display();
    ind_copy.set(ind);
... ind_copy.display();
}
```

Notes

Rules for Overloading

- You can overload based on the number of arguments:

  ```
  void a_function(int a, int b);
  ```

  is different from

  ```
  void a_function(int a, int b, int c);
  ```

  - The compiler determines which one you mean by how many arguments you pass it.

- You can overload based on the types of the arguments.

  ```
  void b_function(int a);
  ```

  is different from

  ```
  void b_function(Date d);
  ```

  - The compiler determines which one you are calling by the types of the arguments you pass it.

- You *cannot* overload based on the types of the return values.

  ```
  int c_function(void);
  ```

  is the same as

  ```
  float c_function(void);
  ```

  - The compiler determines which function to call before it inspects return types.
  - This results in a compile error.

Notes

Overloading Based on Constness

- You can overload based on whether a member function is `const` or not.

  ```
  void Date::display(void);
  ```

  would be different from

  ```
  void Date::display(void) const;
  ```

  - It calls the `const` version if you invoke the method on a `const` object.
  - This allows slightly different behavior based on whether the invoking object is `const` or not.
    - This concept is especially useful with overloaded operators, which we'll cover later.

- One issue worth noting here: During lab, you changed your `Document` class's `virtual void display()` method to be `const`.
  - Because you can overload based on `const`, you must change derived class's `display` method to be `const` also.
    - Otherwise, you're talking about two different methods!
    - Since `display()` is pure `virtual` in the base class, it must be overridden in the derived classes.
    - Therefore, unless you make it `const` in the derived classes, all the derived classes are still abstract!

Notes

Date.h

```
#ifndef _DATE_H_
#define _DATE_H_ 1

class Date {
public:
   Date(void);     // default ctor
   ~Date(void); // dtor
   Date(const Date& d);
   Date(int m, int d, int y=1970);

   void set(int m, int d, int y);
   void set(const Date &dt);

   void display(void) const;
   void increment(void);
   void decrement(void);
private:
   int day;
   int month;
   int year;
};

#endif // #ifndef _DATE_H_
```

No type checking is done by the compiler to determine which arguments have been provided and which arguments should use the defaults. When the function is defined/declared, any default arguments must be specified from right to left. When the function is invoked, any missing arguments are assumed to be the right-most arguments. For Example:

```
class Date {
...
     Date(int m=1, int d=1, int y=1970);
...
};

main()
{
   Date somedate(11);
}
```

The object `somedate` is using the default arguments for setting the `day` and `year`.

Default Arguments

- There are times when you want one overloaded function to be just like another, only not specifying one of the arguments.
  - For instance, we could have a two-`int` constructor for `Date` that took a month and day, with the year set to a default value.

    ```
    Date::Date(int m, int d):
    month(m), day(d), year(1970) { }
    ```

  - But this causes redundant code.
- In C++, you can specify default values for arguments in the declaration (class definition).

  ```
  class Date {
  ...
  public:
       Date(int m, int d, int y=1970);
  ...
  };
  ```

  - No code changes are necessary, if the user code passes in two `int`s, the third (`y`) will be assigned the default value.
  - Default argument values must be specified starting with the last argument and working left.
    - You cannot skip an argument.

    ```
    Date(int m=1, int d=1, int y=1970); // ok
    Date(int m=1, int d, int y=1970);  // illegal
    Date(int m=1, int d=1, int y); // illegal
    ```

Notes

main.cpp

```
...
#include "Date.h"
...
main()
{
...
    Date today(12,20);
    today.display();
}
```

Investigate:

Note the ambiguity in the following code:

inv_1.cpp

```
#include <iostream>

using namespace std ;

void print_number(float fl)
{
   cout << fl << endl;
}

void print_number(double dbl)
{
   cout << dbl << endl;
}

void main(void)
{
   int x=20;
   print_number(x);
}
```

The integer `x` can be converted to either a `float` or a `double`.
How can the bold line be modified to eliminate the ambiguity?

Invoking Functions with Default Arguments

- Beware of the potential for ambiguities in calling functions that have default arguments.
  - ▫ If you have overloaded a function name and you're using default arguments, you can get into trouble.

```
class Date {
...
public:
    Date(int m, int d, int y=1998);
    Date(int m, int d);
...
};

void main()
{
    Date today(10, 25); // Which one is
                        // called?
...
}
```

  - ▫ The answer is neither; the above will result in a compile error because the compiler doesn't know what you mean.
- Otherwise, call the functions or methods normally, omitting any arguments when you want to use their default value.

Notes

Labs

1. Write a `set()` method for `Time` that takes a `Time` object as its argument.

2. Add a `set()` method to your `String` class that takes a `String` object. Why don't we need to overload the `set()` method with one that takes a `char` array? Be careful with this function, there are memory issues involved . . .

3. Create two `set()` methods for `Report`, one that takes a `Report` object as its only argument, and another that takes three arguments: a `Date` object, a `Time` object, and a `String` object.

4. Add two `set()` methods to `Bugreport`, one that takes a `Bugreport` object, and one that takes a `Date` object, a `Time` object, a `String` object, and one `int` (the severity).

5. Make two `set()` methods for the `Bugfix` class, one whose argument is a `Bugfix` object, the other taking three arguments: the fix date, the author of the fix, and a pointer to the `Bugreport` object that shows which bug was fixed.

6. Write two `set()` methods for `ExternalBugreport`, one that takes an `ExternalBugreport` object, another that takes a `Date` object, a `Time` object, a `String` object, an `int` (severity), two more `String` objects, and a status.

Notes