

Preparatory Course in Computing

Summer, 2020

(Lightning) Review of C++

©Prof. R.S. Sreenivas

We will cover the basics of C++ in this lesson. Most of the examples are from [1]. My objective is to bring everyone to a level of comfort and familiarity with C++ so that you can comfortably deal with the *Financial Computing Case Studies* that form the bulk of this course. If you are familiar with C++, this lesson should be a breeze. If you are not familiar with C++, then you will learn enough to comfortably execute the programming assignments. If you are interested in a full-blown course on the nitty-gritty of C++, you might want to take a course with the CS department at a later date.

1 Basics, Global and Local Variables, Typecasting, Scope

Unlike C where there are only two places where you can declare variables – *globally* before a function name, and *locally* after the opening “{” within a block, in C++ you can declare local variables almost anywhere in the program *as long as you declare them before you use them*. The common use of declaring the variables close to where they are being used is within loops. For example

```
for (int i = 0; i < 100; i++)
```

If you had a variable `i` that was declared as an integer, but it is to be used to compute the value of `factor`, which is a float, then in C you would have a line like

```
factor = (float) i * 2.5;
/* typecasts i to floating-point */
```

In C++ you would write it as

```
factor = float (i) * 2.5;
// typecasts i to floating-point
```

Keep in mind that `float` is not a function call (with `i` as input). All typecasts are operators that have precedence over other operators (like `+`, `&&` etc etc).

When you declare a local variable, whether at the top of the block or in the middle, that variable can be used only *within* the block in which it is defined. Consider the C++ program shown in figure 1.

We turn our attention to the *scope resolution* operator “`::`” in C++. If you write a C program with two or more variables of the same name, only the latest/inner-most variable is visible at any one time. C++ uses the *scope resolution* operator to resolve scope conflicts between variable with the same name. First, let us look at the program in figure 2 where we do not use the scope resolution operator, and the program prints the innermost value of `avar`.

```

// Scope Illustration
// Declares variables within two different places in same block
#include <iostream>
using namespace std;

int main()
{
    int var1 = 5; //visible till the end of main()'s block

    for (int var2 = 0; var2 < 10; var2 = var2 + 3) // var2 is now visible
    {
        int var3 = 15;
        cout << "var1_=" << var1 << ",_var2_="
        << var2 << "_and_var3_=" << var3 << endl;
    } // var3 loses visibility here
    // var1 and var2 are still visible here
    // cout << "var3 =" << var3 << endl;
    // remove the comment on the 2nd cout & get a compile-error
    // as var3 is out of scope. Try it with var2 and var1 and
    // see what happens... should give you a clear idea of scope.
}

```

Figure 1: varscope.cpp: Illustration of Scope in C++. If you removed the comments on the second cout statement, you will get a compile-error.

```

// Scope Resolution Operator Illustration
// Four variables with the same name but different scope
// Prof. Sreenivas for IE523: Financial Computing
#include <iostream>
using namespace std;

int avar1 = 5; // 1st declared globally
int main()
{
    int avar2 = 10; // 2nd declaration as an (outer) local variable
    {
        int avar3 = 20; // 3rd declaration as local variable
        {
            int avar4 = 30; // Yet another declaration
            cout << "avar1_=" << avar1 << ",_avar2_=" << avar2 <<
                ",_avar3_=" << avar3 << ",_avar4_=" << avar4 << endl;
            // three print statements... let's see what happens
        } // innermost avar loses scope
        cout << "avar1_=" << avar1 << ",_avar2_=" << avar2 <<
            ",_avar3_=" << avar3 << endl;
    } // 2nd avar loses scope
    cout << "avar1_=" << avar1 << ",_avar2_=" << avar2 << endl;
} // 1st (global) avar loses scope

```

Figure 2: `avar1.cpp`: Part One of the Illustration of the Scope Resolution Operator in C++.

Some compilers might give you a warning message that could clue you in about the confusion of scope. The output will be

```
avar = 30, avar = 30, avar = 30
```

Now, look at the code in figure 3. You will get the following output on this code

```
avar = 30, avar = 5, avar = 5.
```

To my knowledge, there is no way to access the second and third `avar` variables, only the innermost and global `avar`'s are accessible. There are other more important reasons for the scope resolution operator `::` that we will see later, but the code shown in figure 4, should give some idea of its other use.

This one is an old classic – the `#define` statement can be used to define macros as well as constants in C++ (just as in C). But you have to be careful with its use. Take a look at the program listed in figure 5. When run on the input 3, it produces the output

```

Please type a number 3
The cube of 3 is 27
The cube of 4 is 10.

```

```

// Scope Resolution Operator Illustration
// Four variables with the same name but different scope
#include <iostream>
using namespace std;

int avar = 5; // 1st declared globally
int main()
{
    int avar = 10; // 2nd declaration as an (outer) local variable
    {
        int avar = 20; // 3rd declaration as local variable
        {
            int avar = 30; // Yet another declaration
            cout << "avar_1st_=" << avar << ",_avar_2nd_=" << ::avar <<
                ",_avar_3rd_=" << ::avar << endl;
            // three print statements... let's see what happens
        } // innermost avar loses scope
    } // 2nd avar loses scope
} // 1st (global) avar loses scope

```

Figure 3: `avar2.cpp`: Part Two of the Illustration of the Scope Resolution Operator in C++.

```

// Scope Resolution Operator Illustration
// Four variables with the same name but different scope
#include <iostream>
//using namespace std;

int avar = 5; // 1st declared globally
int main()
{
    int avar = 10; // 2nd declaration as an (outer) local variable
    {
        int avar = 20; // 3rd declaration as local variable
        {
            int avar = 30; // Yet another declaration
            std::cout << "avar_=" << avar << ",_avar_=" << ::avar <<
                ",_avar_=" << ::avar << std::endl;
            // three print statements... let's see what happens
        } // innermost avar loses scope
    } // 2nd avar loses scope
} // 1st (global) avar loses scope

```

Figure 4: `avar3.cpp`: Part Three of the Illustration of the Scope Resolution Operator in C++.

```

// Inline function definition that computes the cube of a number
#include <iostream>
using namespace std;

#define CUBE(x) x*x*x

int main()
{
    int number, cube_of_number;
    cout << "Please_type_a_number_";
    cin >> number;
    cube_of_number = CUBE(number);
    cout << "The_cube_of_" << number << "_is_" << cube_of_number << endl;
    // now for some weird stuff
    cube_of_number = CUBE(number+1); //cube of the next number
    cout << "The_cube_of_" << number+1 << "_is_" << cube_of_number << endl;
}

```

Figure 5: macro1.cpp: A word of caution about using inline functions.

Clearly, something is wrong as $4^3 \neq 10$. This happens because the `#define` preprocessor directives are essentially asking the compiler to basically do a “find-and-replace” job whenever `CUBE` is encountered in the code. So, `CUBE(number+1)` becomes

$$\text{cube_of_number} = \text{number} + 1 * \text{number} + 1 * \text{number} + 1$$

and when `number = 3`, the above expression evaluates to $3 + 1 * 3 + 1 * 3 + 1 = 3 + 3 + 3 + 1 = 10$ – keep in mind that multiplication has precedence over addition in almost all programming languages. A bunch of correctly placed parenthesis would take care of this problem – as shown in figure 6. We have the following output for this example.

```

Please type a number 3
The cube of 3 is 27
The cube of 4 is 64.

```

Inline function calls like this are usually done when a function is repeatedly used and we want to avoid the function call overhead. To turn a function into an inline function, you could place the `inline` keyword before the function name (cf. figure 7, for example). Some compilers might just ignore the inline-request and proceed with the compilation as though these are regular function calls. You want to do this only with short/small function calls. Most compilers will ignore inline-requests if the code exceeds some length.

Regarding input/output, you must have had some idea of how it is done in the examples we have seen so far. The *objects* that are used in this context are

```

// Inline function definition that computes the cube of a number
#include <iostream>
using namespace std;

#define CUBE(x) (x)*(x)*(x)

int main()
{
    int number, cube_of_number;
    cout << "Please_type_a_number_";
    cin >> number;
    cube_of_number = CUBE(number);
    cout << "The_cube_of_" << number << "_is_" << cube_of_number << endl;
    // now for some weird stuff
    cube_of_number = CUBE(number+1); //cube of the next number
    cout << "The_cube_of_" << number+1 << "_is_" << cube_of_number << endl;
}

```

Figure 6: macro2.cpp: The corrected version of the inline function example shown in figure 5.

```

// Inline function definition that computes the cube of a number
#include <iostream>
using namespace std;

inline int CUBE(int x)
{
    return x*x*x;
}

int main()
{
    int number, cube_of_number;
    cout << "Please_type_a_number_";
    cin >> number;
    cube_of_number = CUBE(number);
    cout << "The_cube_of_" << number << "_is_" << cube_of_number << endl;
    // now for some weird stuff
    cube_of_number = CUBE(number+1); //cube of the next number
    cout << "The_cube_of_" << number+1 << "_is_" << cube_of_number << endl;
}

```

Figure 7: macro3.cpp: Inline function illustration.

`cin`, `cout`, `cerr` and `clog`. The `endl` keyword sends a newline to the screen and flushes the relevant buffers. As with many objects in *object oriented programming*, you really do not need to know how these objects work. These objects deal with writing and reading from the screen/keyboard. We will discuss reading and writing data to files at a later date.

2 Pointers, References and Memory Allocation in C++

I am going to assume everyone is familiar with what a *pointer* is. If you need a refresher – you can view the memory of the computer as a bunch of locations (or cells) of a byte each. When you declare a variable, a prescribed amount of memory is set aside for the variable at some memory location. We really do not need the specific address of the location where this variable is stored, but there are many reasons why we might want to manipulate the address (and contents) of memory locations in our code. If we have a line like

```
alpha = & beta,
```

then the value of `alpha` is the “address of” `beta`. If you have a line like

```
gamma = * delta,
```

then `gamma` is equal to the value pointed by the address `delta`.

2.1 References

References are new kinds of C++ variables. We could spend a lot of time on this, and if pointers are new to you, what ever I say will sound alien to you. Instead, here is a low-brow way of telling you why these are new variables – in C the “&” operator stands for “address of,” and it usually appears on the right-hand-side of an assignment. In C++ this operator is permitted to be on the left-hand-side of an assignment. A C++ statement like

```
int & ref1 = ivar
```

is essentially saying the address of `ref1` and that of `ivar` are the same – which is pretty much saying that `ref1` and `ivar` are proxies/aliases of each other. A careful study of the code shown in figure 8 should tell you more about the use of *references* in C++. You should get the output shown below.

ivar1 is 10	After ivar1++:	After iref = ivar2:
ivar2 is 20	ivar1 is now 11	ivar1 is now 20
iref 10	ivar2 is now 20	ivar2 is now 20
	iref is now 11	iref is now 20
After iref++:		
ivar1 is now 21		
ivar2 is now 20		
iref is now 21		

```

// Illustrative code about references in C++
#include <iostream>
using namespace std;

int main()
{
    int ivar1 = 10; // ivar1 is declared integer & assigned a value
    int ivar2 = 20; // same thing with ivar2
    int & iref = ivar1; // iref is a refernce to ivar1

    cout << "ivar1_is_" << ivar1 << endl;
    cout << "ivar2_is_" << ivar2 << endl;
    cout << "iref_" << iref << endl << endl;

    ivar1++;
    cout << "After_ivar1++:" << endl;
    cout << "ivar1_is_now_" << ivar1 << endl;
    cout << "ivar2_is_now_" << ivar2 << endl;
    cout << "iref_is_now_" << iref << endl << endl;

    iref = ivar2; // iref and ivar1 *BOTH* contain value of ivar2
    cout << "After_iref_=ivar2:" << endl;
    cout << "ivar1_is_now_" << ivar1 << endl;
    cout << "ivar2_is_now_" << ivar2 << endl;
    cout << "iref_is_now_" << iref << endl << endl;

    iref++;
    cout << "After_iref++:" << endl;
    cout << "ivar1_is_now_" << ivar1 << endl;
    cout << "ivar2_is_now_" << ivar2 << endl;
    cout << "iref_is_now_" << iref << endl << endl;
}

```

Figure 8: ref.cpp: Illustration of *Reference* variables in C++.

2.2 Structures and Enumerated Data Types

In C, you would declare a structure with the following statements

```
struct nameadr {  
    char name[30];  
    int age;  
    float salary;  
};
```

To define a structure variable, you must list the **struct** keyword, along with the structure tag **nameadr**, as follows

```
struct nameadr person; /* A single struct variable */  
struct nameadr people[100]; /* An array of 100 struct variables */
```

Because C has no built-in data type **nameadr**, the user must define one – hence the term *user-defined data type*. Some C programmers use the **typedef** keyword to define the new data type to the compiler as follows.

```
typedef struct {  
    char name[30];  
    int age;  
    float salary;  
} nameadr;
```

Once you define the type to C, you can define the structure variables without **struct** as

```
nameadr person; /* A single struct variable */  
nameadr people[100]; /* An array of 100 struct variables */
```

In C++, once you define a special data type such as a structure (or union or an enumerated data type), C++ remembers that data type. You do not need the **struct** keyword again. You could do something like this in C++:

```
struct nameadr {  
    char name[30];  
    int age;  
    float salary;  
};  
nameadr person; /* A single struct variable */  
nameadr people[100]; /* An array of 100 struct variables */
```

2.3 Allocating Memory in C++

In C you might have used **malloc** and **free**. In C++ you will use **new** and **free** instead. When you allocate free memory in any program you should take care of any *exception handling* that might result (for eg. if there is insufficient memory, etc.). This is a very important issue if your program is meant to run in a multitasking environment like a personal computer. The amount of free

```

// Illustrative code about exception handling in C++
#include <iostream>
#include <stdlib.h>
#include <new>

using namespace std;

extern void (* set_new_handler(void (*memory_err())()))();
// declares a pointer to a function in new.h
// that takes a function pointer as its argument

void memory_err() {
    cout << "A memory allocation error occurred... quitting\n";
    exit(1); // quits the program
}

int main()
{
    set_new_handler(memory_err); // setting up the exception handler

    float *fp1 = new float [10]; // you should have no trouble here
    std::cout << "First allocation worked." << std::endl;

    // let's try and trip the exception handling
    float *fp2 = new float [999909999999999999];
    std::cout << "Second allocation worked." << std::endl;

    delete fp1;
    delete fp2; // cleaning up
}

```

Figure 9: except.cpp: Illustration of exception handling in C++.

memory that is available depends on the other programs that are currently being run, and without exception handling you can have unpredictable behavior when you run your code. For this, it is useful to know that C++ calls the function `set_new_handler()` every time `new` fails. The program listed in figure 9 should tell you how exception handling for memory allocation errors works. Here is what I get when I run this program on my MacBook Air.

```

First allocation worked.
ref(5687) malloc: *** mmap(size=4000002048) failed (error code=12)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
A memory allocation error occurred... quitting

```

2.4 C++ Functions

There are three ways to pass values from one function to another in C++. C++ shares two of these three methods with C. Both C and C++ pass values by *value* and *by address*. In C++ you can also pass values by *reference*.

When you pass arguments from one function to another by *value* – you are essentially copying the value of the variable from the calling function to the variable in the receiving function. When the receiving function modifies the value, it modifies only its copy. The value of the calling function is left unchanged. This has its advantages in that it protects the data’s original contents. Because of the copies that are being made, too much of this can result in memory shortage problems. Another reason why this might not be attractive is the time overhead it needs to make copies.

When you pass a pointer from one function to another, or precede the argument with the “address of” operator – `&` – you are passing the address of the argument instead of a copy of the argument’s value. In this case, any changes in the receiving function would result in a similar change in the a calling function too.

The code shown in figure 10 illustrates these two methods of passing arguments. The resulting output is shown below.

```
Before function call fun1(), val is 15
Inside fun1(), val is 30
After function call fun1(), val is 15
Inside fun2(), val is 30
After function call fun2(), val is 30
```

Now, for something unique to C++ – *passing by reference* – with this you can pass scalar arguments by address without the messy `&` and `*` notation. When you pass a reference variable, you pass an alias/proxy to the variable. We saw that a reference variable is a special type of pointer. It contains the same memory location as the variable to which it points. Loosely speaking, when you pass a reference variable, you pass not only the address, but the value itself. When you pass a variable by reference, the receiving function gets an alias of the calling function’s variable, which is actually the same value in every respect, but with a different name. There is no need for the extra `&`’s and `*`’s, as you would have when you pass by address. Figure 11 contains an illustrative example. A sample output is presented below.

```
How old are you? 45
Are you really 45 years old?
```

The example shown in figure 12 the potential for default arguments in C++, and it should be self-explanatory. The resulting output is shown below.

```
i: 5 j: 40034 x: 10.25 ch: Z d: 4.323
i: 2 j: 40034 x: 10.25 ch: Z d: 4.323
i: 2 j: 75037 x: 10.25 ch: Z d: 4.323
```

```

// Illustrative code about passing by value and address in C++
// Same as what you might see in C
#include <iostream>
using namespace std;

int fun1(int val1) // receives val by value
{
    //cout << "Inside fun1(), &val1 = " << &val1 << endl;
    //cout << "Before calling fun1(), val1 is " << val1 << endl;
    val1 += 15; // adds 15 to the value
    //cout << "After calling fun1(), val1 is " << val1 << endl;
    cout << "Inside fun1 the address of val1 is " << &val1 << endl;
    return (val1);
}

void fun2(int *val2) // receives val by address
{
    cout << "What is passed to fun2 is : " << val2 << endl;
    *val2 += 15; // adds 15 to the value
    cout << "Inside fun2(), val2 is " << *val2 << endl;
}

void fun3(int &val3) // receives val by reference
{
    cout << "Inside fun3(), &val3 = " << &val3 << endl;
    cout << "The value of val3 (ie. whats-in-the-box of val3) = ";
    cout << val3 << endl;
    val3 += 15;
    cout << "Inside fun3(), val3 is " << val3 << endl;
}

int main()
{
    int val = 15;

    /*
    cout << "Before function call fun1, &val = " << &val
    << endl;
    cout << "Before function call fun1(), val is " << val << endl;
    cout << "This stuff is newly added " << fun1(val) << endl; // pass by value
    cout << "After function call fun1, &val = " << &val
    << endl;
    cout << "After function call fun1(), val is " << val << endl;
    */

    /*
    cout << "Before function call fun2, &val = " << &val
    << endl;
    cout << "Before function call fun2(), val is " << val << endl;
    fun2(&val); // pass by address, this time
    cout << "After function call fun2(), val is " << val << endl;
    */

    cout << endl;
    cout << "Before function call fun3, &val = " << &val
    << endl;
    cout << "Before function call fun3(), val is " << val << endl;
    fun3(val); // pass by reference, this time
    cout << "After function call fun3, &val = " << &val
    << endl;

```

```

// Illustrative code about passing by reference in C++
#include <iostream>
using namespace std;

void get_age(int &age) // age is a reference to main()'s age
{
    cout << "How_old_are_you?_";
    cin >> age;
}

int main()
{
    int age;
    get_age(age); // passes by reference
    cout << "Are_you_really_" << age << "_years_old?" << endl;
}

```

Figure 11: byref.cpp: Illustration of passing by *reference* in C++.

```

i: 2 j: 75037 x: 35.88 ch: Z d: 4.323
i: 2 j: 75037 x: 35.88 ch: G d: 4.323
i: 2 j: 75037 x: 35.88 ch: G d: 0.0023

```

C++ also lets you have more than one function with the same name – that is, it permits *overloading* – however, each overloaded function must differ in their arguments. The code shown in figure 13 illustrates this concept.

```

C++ knows this is a string argument: Bunga Bunga Bunga
C++ knows this is a int argument: 123456
C++ knows this is a float argument: 3.14159

```

3 Objects and Classes

I am going to steer clear of a lot of (relevant and important) material about what an *object* and a *class* is etc etc. These topics are to be treated eloquently and carefully, but we really do not have the time – we need to get going with the financial case studies in a week. So here goes – a *class* in C++ is best thought of as a *structure* in C with a whole bunch of additional/extra stuff with it. We are going to list the “extra stuff” now, and they will be discussed in detail in subsequent text.

- *Extra Stuff #1: Public and Private members.*
- *Extra Stuff #2: Member functions as a part of the structure.*
- *Extra Stuff #3: constructor and destructor functions.*

```

// Illustrative code about default argument list for a function with
// several parameters
#include <iostream>
#include <iomanip>
using namespace std;

void default_function(int i=5, long j = 40034, float x = 10.25,
                     char ch='Z', double d = 4.3234); // Prototype

void default_function(int i, long j, float x, char ch, double d)
{
    cout << setprecision(4) << "i:_" << i << "_j:_" << j;
    cout << "_x:_" << x << "_ch:_" << ch << "_d:_" << d << endl;
    return;
}

int main()
{
    default_function(); // all defaults used
    default_function(2); // first default overridden
    default_function(2, 75037); // 1st & 2nd overridden
    default_function(2, 75037, 35.88); // 1st, 2nd and 3rd overridden
    default_function(2, 75037, 35.88, 'G'); // only the 5th value is default
    default_function(2, 75037, 35.88, 'G', 0.0023); // no defaulting
}

```

Figure 12: default_arg.cpp: Illustration of passing by *reference* in C++.

```

// Illustrative code about function overloading in C++
// three different functions — same name — different types of inputs
#include <iostream>
#include <iomanip>
using namespace std;

void output(char []); // Prototype for the character input version
void output(int i); // prototype for the integer input version
void output(float x); // prototype for the float input version

void output(char name[])
{
    cout << "C++ knows this is a string argument:" << name << endl;
    return;
}

void output(int ivalue)
{
    cout << "C++ knows this is a int argument:" << ivalue << endl;
    return;
}

void output(float fvalue)
{
    cout << "C++ knows this is a float argument:" << fvalue << endl;
    return;
}

int main()
{
    char name[] = "Bunga_Bunga_Bunga";
    int ivalue = 123456;
    float fvalue = 3.141592654;

    output(name); // C++ knows what to do
    output(ivalue);
    output(fvalue);
}

```

Figure 13: overload.cpp: Illustration of function *overloading* in C++.

For starters, as a programming construct you would declare *classes* in C++ just as you would do with structures in C, it is just that you substitute the term **struct** with **class**. Now for a single sentence on the term *object* – *objects* are essentially variables (i.e. a defined region in memory), which are instances of a *class*. Suppose you define a C++ class called **Date** as

```
class Date {
    int day;
    int month;
    int year;
}
main ()
{
    Date today; //defines a new class variable;
    blah blah
}
```

In C++ you will run into a big problem – you will never be able to assign the values of the members of this class. For instance, in C you might have a couple of lines like:

```
today.day = 25;
today.month = 8;
today.year = 2010;
```

This will not work in C++ – and this is because of the notion of *private* and *public* members of a class. Unless you declare them **public**, all members of a class are assumed to be **private** by default. There is no way (other than using something called *member functions*, to be described later) for any part of the code to access the **private** members. As you program more, you will begin to realize that this is a good thing, although it might seem a little weird at first.

Some books refer to *member functions* as *methods*. I am going to steer clear of all this – to us, a **class** in C++ is something like a **struct** in C, but it has **private** and **public** members, along with a member function that can be used to access **private** members (and then some). Many books refer to this whole-package as *encapsulation*. There is more to this as a design-philosophy/design-concept but this level of familiarity will suffice for the moment.

Bringing up the rear, are *constructor* and *destructor* functions. They are special member functions (with a strict naming syntax) that initialize and clean up class objects. For the present, you can think of a constructor as a member function that makes sure objects are defined with initial values without worrying about what is **public** and **private** etc etc. We will now look at several code samples where these concepts are reinforced.

3.1 Illustration 1: The Date Class

The code shown in figure 14 illustrates the concepts introduced earlier. The comments alongside the code should tell you what each line is about. Send me an e-mail if you are lost or need help. After a fashion, here is what is going on with this code – we defined a class called `Date`. Its private members are `day`, `month` and `year`. We then present the compiler with the prototypes of the private functions `test_day()`, `test_month()` and `test_year()` – each of which returns a 1 (0) if the the member they are meant to test is legitimate (incorrect). We just put an unused integer called `bonus_flag` as the only public member of this class. The public member functions (or “methods”, if you wish) get the day, month and year. The last public member function sets the day, month and year after testing them using the private functions. The `main()` function takes the user input and assigns the day, month and year after checking for correctness.

You can imagine adding other member functions – like comparing dates, figuring out the number of days between two legitimate `day/month/year` inputs, etc etc. As these functions are added, the `main()` function does not have to change, if all you cared about was what you had originally written it to do. Usually, the class definitions are included in a header file, say `Date.h`, that is included in the C++ code (using a `#include "Date.h"` command in the first few lines of the program). Commercial classes come with huge header files. They usually have detailed documentation that describes the class and how to use it.

```
What is today's date, month and year (format: d m y)? 24 8 2010
The date is set to 8/24/2010
What is today's date, month and year (format: d m y)? 1 100 1900
You did not enter a correct date.
```

3.2 Illustration 2: The PaySave Class

The following program is meant for use in a payroll department to track employee benefit savings plan. We define a class called `PaySave` in this program. The program tracks data for employee accounts, tracking interest rates and employer deposits. The program also requires that the account balance be printed and that each month the account's interest be credited. The code shown in figure 15, along with the comments, should be self-explanatory. A sample output is presented below

```
The Boss's payroll investment balance information:
Balance: $0.00
Monthly Rate: 0.30%

The Worker's payroll investment balance information:
Balance: $0.00
Monthly Rate: 0.20%
```

```

// Date Class Illustration
#include <iostream>
using namespace std;

class Date
{
    int day; //private data members
    int month;
    int year;
    int test_day (const int d) { // private member functions
        return (d > 31) ? 0:1;
    }
    int test_month (const int m) {
        return (m > 12) ? 0:1;
    }
    int test_year (const int y) {
        return (y < 2000) ? 0:1;
    }
public:
    int bonus_flag; //public bonus date flag
    int get_day (void)
    { // public member function
        return (day);
    }
    int get_month (void)
    {
        return (month);
    }
    int get_year (void)
    {
        return (year);
    }
    int set(const int d, const int m, const int y)
    {
        if (test_day(d) && test_month(m) && test_year(y))
        {
            day = d; // all numbers are legit; assign away!
            month = m;
            year = y;
            return (1);
        }
        else
            return (0); // Bad date!
    }
};

int main()
{
    Date today; //defines a class variable
    int m, d, y; //user's input values

    cout << "What is today's date, month and year (format: d_m_y)? ";
    cin >> d >> m >> y; // notice the order!

    //set the date class value & print if OK
    if (today.set(d, m, y)) {
        cout << "The date is set to " << today.get_month();
        cout << "/" << today.get_day() << "/";
        cout << today.get_year() << endl;
    }
    else {
        cout << "You did not enter a correct date." << endl;
    }
}

```

Figure 14: date.cpp: Illustration of C++ classes and related miscellanea.

At the end of the month 1, the boss's balance is \$250.00
At the end of the month 1, the worker's balance is \$100.00

At the end of the month 2, the boss's balance is \$500.75
At the end of the month 2, the worker's balance is \$200.20

At the end of the month 3, the boss's balance is \$752.25
At the end of the month 3, the worker's balance is \$300.60

The well-worn academic illustration of object oriented programming compares a C++ program to the components of a stereo system. The CD player might be of a particular brand, the receiver a different brand, the DVD player yet another brand, and finally the speakers might be all together different make than the rest. But, when they are put together, they form a single stereo system in your entertainment room. You do not need the nitty-gritty details of how each component works to put together something that suits your taste. All you need to know is how these components should be connected together. The same holds for C++ class libraries. When you use a class, whether you purchase it from a software vendor or you may write it yourself, you really do not need to know the inner-workings of the class code. You only need to know how to use the class – its member functions, and the arguments they take – the names of the public data members – etc. You do not need the code for how these things work. This, in a nutshell is the essence of object-oriented programming languages like C++.

3.3 Polymorphism

This is big word that is bandied around a lot in object-oriented programming. To understand what it means, you should look at the code in figure 15. In that we used the same member functions – like `get_rate()` and `month_end()`, for example – on two different objects – `boss` and `worker`. That is, we used `boss.month_end()`; and `worker.month_end()`. We have an instance of the same function acting on two different objects.

Granted, these two objects `boss` and `worker` had sort of similar looking “formats.” But, they did not have to be similar looking for this to work. The fact that you can apply the same function across different objects is known as *polymorphism*. The real-life analogy that is often times given in this context is that while a box and a door are different objects, the act of opening them is sort of – in a suitably abstract sense – similar. The “opening” is a common function that is applied to different objects in this case – and this is *polymorphism*. In a programming context it can clear a lot of clutter in the code.

```

// PaySave Class Illustration
#include <iostream>
#include <iomanip>

using namespace std;

class PaySave {
    float rate; // Rate savings plan currently earns
    float month_company; // monthly amount company contributes
    float balance; // balance for this employee
public:
    float get_rate (void) {
        return (rate);
    }
    float get_month_company (void) {
        return (month_company);
    }
    float get_balance (void) {
        return (balance);
    }
    void set_rate (const float r); // sets rate
    void set_company (const float m); // sets company's contribution
    void set_balance (const float b); // sets beginning balance
    void employee_deposit (const float employee_amount); // Adds deposit
    double employee_withdrawal (const float employee_amount); // subtracts withdrawal
    void month_end (void); // updates balance with interest rate
};

// Code for member functions defined outside the class
void PaySave::set_rate(const float r) { // sets the rate
    rate = (r > 0.0) ? r : 0.0; // makes sure rate is not negative
}

void PaySave::set_company(const float m) { // sets company's month-end contribution
    month_company = (m > 0.0) ? m : 0.0;
}

void PaySave::set_balance(const float b) { // sets initial balance
    balance = b;
}

void PaySave::month_end(void) { // Updates balance with month-end figures
    balance += ((rate * balance) + month_company);
}

int main()
{
    PaySave boss, worker; //two class variables

    boss.set_rate(0.003); //Boss' initial data
    boss.set_company(250.0);
    boss.set_balance(0.0);

    worker.set_rate(0.002); //Yes, rate is smaller :- )
    worker.set_company(100.0);
    worker.set_balance(0.0);

    cout << setiosflags(ios::fixed) << setprecision(2); // sets outputs for two decimals
    cout << "The_Boss's_payroll_investment_balance_information:_ " << endl;
    cout << "Balance:_ $" << boss.get_balance() << endl;
    cout << "Monthly_Rate:_ " << (boss.get_rate() * 100.0) << "%" << endl << endl;

    cout << "The_Worker's_payroll_investment_balance_information:_ " << endl;
    cout << "Balance:_ $" << worker.get_balance() << endl;
    cout << "Monthly_Rate:_ " << (worker.get_rate() * 100.0) << "%" << endl << endl;

    // update for end-of-month values for five months
    for (int i = 0; i < 3; i++) {
        cout << endl;
        boss.month_end();
        worker.month_end();

        cout << "At_the_end_of_the_month_" << i+1 << ",_the_boss's_balance_is_$";
        cout << boss.get_balance() << endl;
        cout << "At_the_end_of_the_month_" << i+1 << ",_the_worker's_balance_is_$";
        cout << worker.get_balance() << endl;
    }
}

```

Figure 15: paysave.cpp: Yet another Illustration of C++ classes and related miscellanea.

3.4 Constructor and Destructor Functions

Let us wrap this lesson up with a quick overview of *constructor* and *destructor* functions. These are special member functions that initialize and clean up **class** objects. You list constructor and destructor functions along with the rest of the member functions in a **class**. A class can have as many constructors as there are ways to initialize it. But, you can only have one destructor function. There are some restrictions on how they can be named though – the constructor functions have the same name as the class itself. The destructor function also has the same name as the class, but it is preceded with a “~.”

Constructors are called (unknown to the user) in many ways – when you declare a variable `int i = 10`, a constructor is called that reserves a location that is of the size of an integer and it attaches the name `i` to it, and puts a value of 10 in that location. When a variable goes out of scope, a destructor function that releases the memory location used by the variable.

With a constructor that is defined for a class, the compiler defines and initializes the requisite objects. You never explicitly call the constructor functions in your code. The compiler calls the constructor for you when you define the object. Likewise, the compiler calls an object’s destructor function when the object goes out of scope. The program shown in figure 16 illustrates how this is done. Not much happens in the destructor – except that a message prints and shows that C++ calls the destructor for every array element. The sample output should tell you a lot.

```
c, i, x are A, 25, and 123.4
c, i, x are A, 25, and 123.4
c, i, x are A, 25, and 123.4
Destructor called.
Destructor called.
Destructor called.
```

4 Operators and Friends

There might be a few occasions where member functions among a group of classes might have to access each others private members. The **friend** function in C++ lets you do just that. The **friend** functions might, or might not be member functions of another class. Consider the example shown in figure 17. The function `print_em()` is a **friend** of the two classes **Customer** and **Account**, and consequently it has access to the private members of this class. The output is shown below. You can make an entire class a **friend** of another, and the **friend** relationship among classes is not reflexive (i.e. if **a** is a **friend** of **b**, it does not necessarily mean that **b** is a **friend** of **a**).

You can make an entire class a **friend** of another. As with shared classes, you must declare the nested class before using it inside the other class. Here is an example.

```

// Constructor is called for every allocated array object and the
// destructor is called everytime the objects leave scope.
// The destructor does nothing but print a message for us to see.
#include <iostream>
#include <iomanip>

using namespace std;

class Sample {
    char c;
    int i;
    float x;
public:
    Sample(); // constructor
    ~Sample(); // destructor
    void print_it(void);
};

//Member functions
inline Sample::Sample() {
    c = 'A'; // assigns default values
    i = 25;
    x = 123.45;
}

inline Sample::~~Sample() {
    cout << "Destructor_called." << endl;
}

inline void Sample::print_it(void) {
    cout << setiosflags(ios::fixed) << setprecision(1); // sets outputs for one decimal
    cout << "c,i,x_are_" << c << ",_" << i << ",_and_" << x << endl;
}

int main()
{
    const int count = 3; //just a number

    //This line allocates and initializes each array element
    Sample *sp = new Sample[count];

    for (int i = 0; i < count; i++)
        sp[i].print_it();

    delete [] sp;
}

```

Figure 16: constr_destr.cpp.cpp: Illustration of Constructor and Destructor Functions.

```

// Illustration of friend function
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

class Account; //Let the compiler know that Account class exists

class Customer {
    char name[30];
    int cnum;
    float ytd_balance;
public:
    Customer() {
        strcpy(name, "Joe");
        cnum = 10;
        ytd_balance = 31.96;
    }
    friend void print_em (const Customer c, const Account a); //friend declaration
};

class Account {
    char acctcode[5];
    float ac_balance;
public:
    Account() {
        strcpy(acctcode, "123");
        ac_balance = 43.22;
    }
    friend void print_em (const Customer c, const Account a); //friend declaration
};

void print_em (const Customer c, const Account a) {
    cout << setiosflags(ios::fixed) << setprecision(2); // sets to two decimals
    cout << "Customer:_ " << c.name << ",_Number:_ " << c.cnum;
    cout << ",_YTD_Balance:_ " << c.ytd_balance << endl;
    cout << "Account:_ " << a.acctcode;
    cout << ",_Balance:_ " << a.ac_balance << endl;
};

int main()
{
    Customer c;
    Account a;

    print_em(c, a); // calls the friend function
}

```

Figure 17: friendcpp: Illustration of friend functions in C++.

```

class Parttime; // Declares the nested class
class Employee {

    char name[30];
    int enum;

public:

    friend class Parttime; // Nesting the friend declaration

}; class Parttime {

    int hours;
    float rate;

public:

    void print();

};

```

The nested friend can be private or public.

4.1 Operator Overloading

In a nutshell, operator overloading is the process by which you tell C++ how to apply operators such as the addition sign, +, to your own abstract data types, such as class data. There are some operators in C++ that cannot be overloaded, the compiler would let you know if you did that anyways. I will skip the part where we go through those forbidden things.

4.1.1 Overloading I/O operations

The << operator is already overloaded in C++. It is both a bitwise left-shift operator and when cout appears on the left-hand-side, it is an output operator. If you compiled a C++ code without the #include iostream statement, you would get an error that says something like

```

Undefined symbol 'cout'

```

If you included iostream, it overloads << and defines the cout object, which is an object of the an output stream class called ostream. iostream.h includes an overloaded operator function that looks like –

```

ostream & operator<<(ostream & c, int n);

```

along with several other overloadings that accept floating-points, characters, and all the other built-in and user-defined data types. The one shown above takes two arguments – (1) an output stream, and (2) an integer. It returns a reference to the output stream ostream. Suppose you have a statement like


```
cout << 15 << 25; // outputs two integers to output device
```

First, C++ knows the overloaded << output operator is to be used instead of the bitwise left-shift operator. It then sends the integer 15 to the `cout` object, and the overloaded function returns an output stream object, then the integer 25 is sent to *that* returned `cout` object. This way you can keep “stacking” the commands as needed. If the `operator<<()` did not return a reference to the output stream object, you would be forced to separate the output command into two commands as

```
cout << 15;  
cout << 25;
```

The code shown in figure 18 lets us get a date from the user by a single line

```
cin >> today;
```

and permits us to print the entire `Date` class by a line

```
cout << today;
```

A sample run of this program is shown below.

```
What is the day number (1 to 31)? 27  
What is the month number (1 to 12)? 8  
What is the year number (i.e. 2010)? 2010
```

```
Date: 8/27/2010
```

4.1.2 this Pointer

Now might be the time to review `this` pointers. When something like the `PaySave` class program of the previous lesson calls a member function like

```
boss.set_rate(0.07);
```

and

```
worker.set_company(100.0);
```

C++ actually sends to `set_rate()` and `set_company()` the function calls –

```
set_rate(&boss, 0.07);
```

and

```
set_company(&worker, 100.0);
```

```

// Overloading operators illustration involving the Date class
#include <iostream>
#include <iomanip>

using namespace std;

class Date {
    int day;
    int month;
    int year;
    friend istream & operator>> (istream & in, Date & d);
    friend ostream & operator<< (ostream & out, Date & d);
};

istream & operator>> (istream & in, Date & d)
{
    cout << endl;
    cout << "What_is_the_day_number_(1_to_31)?_";
    in >> d.day;
    cout << "What_is_the_month_number_(1_to_12)?_";
    in >> d.month;
    cout << "What_is_the_year_number_(i.e._2010)?_";
    in >> d.year;
    return in;
};

ostream & operator<< (ostream & out, Date & d)
{
    cout << endl;
    cout << "Date:_ " << d.month << "/" << d.day << "/" << d.year << endl;
    return out;
}

int main()
{
    Date today;
    cin >> today;
    cout << today;

    int x;
    cin >> x;
    cout << endl << x << endl;

    cout << "Testing.." << endl;
}

```

Figure 18: input_output_overloading.cpp: Illustration of Input/Output overloading in C++.

When you call a member function, preceding the function name with the object name, C++ takes the address of that object and passes it to the member function. The receiving member function's parameter list does not look the way it does in the class either. There is an extra parameter to receive the address to the object. Here is how the prototypes for `set_rate()` and `set_company()` looks

```
void set_rate(PaySave * this, const float r);
```

and

```
void set_company(PaySave * this, const double m);
```

C++ calls the hidden object pointer `this` and precedes the data members with a reference to `this`. Suppose you have a member function like

```
void PaySave::set_rate(const float r) {
    rate = (r > 0.0) ? r : 0.0; // ensures against negative rate
}
```

Internally, C++ changes it to

```
void PaySave::set_rate(PaySave *this, const float r) {
    this->rate = (r > 0.0) ? r : 0.0; // ensures against negative
    rate
}
```

Basically, you do not have to bother with the extra argument, as long as you call the function with the correct object, C++ will do the rest by itself.

4.1.3 Overloading Math Operators

You can add two classes if you wish. Keep in mind that the exact process of ‘adding’ can be different for different classes. You can control exactly what got added, and how, by operator overloading. So, if you had an `hourly` employee class and a `salaried` employee class, you could do something like

```
total = hourly + salaried; // Adds two different objects
```

You cannot redefine the way operators work with built-in data types. In other words, you cannot modify the way the “+” sign works with integers. You can only specify how the “+” works with your own data types. You cannot change the operator precedence either. If you overloaded the “*” operator to work on your own class data, the overloaded “*” operator has the same precedence as the regular multiplication operator. An operator overloaded function looks just like any other C++ function, with one exception: the name of the function must be `operator`, followed by the operator you want to overload.

Consider the code shown in figure 19. We have a list of five values, and the +, - and * operators are defined on a term-by-term basis as shown in the code. The resulting output is

The result: [60, 60, 60, 60, 60]
The result: [40, 40, 40, 40, 40]
The result: [500, 500, 500, 500, 500]

4.1.4 Some Additional Orphaned-Material

There are a few additional things that you need to know – at least when it comes to being cogent when you are asked a C++ related question in an interview. I will review them below

4.1.5 static and auto variables

A **static** variable is one that does not lose its value when it goes out of scope. In contrast, an **auto** variable disappears when it goes out of scope. A global variable is pretty much like a **static** variable, it is just that they do not need the **static** keyword. This is demonstrated by the code in figure 20, which results in the following output when executed.

Calling count1() three times.
count1() called 1 time(s) so far.
count1() called 1 time(s) so far.
count1() called 1 time(s) so far.

Calling count2() three times.
count2() called 1 time(s) so far.
count2() called 2 time(s) so far.
count2() called 3 time(s) so far.

Calling count3() three times.
count3() called 1 time(s) so far.
count3() called 2 time(s) so far.
count3() called 3 time(s) so far.

5 Inheritance

There is a lot that can be said about *inheritance*, and as with everything else in this two-week review, I am going to skip a lot of the verbiage, and stick to the core-issues.

C++ utilizes a hierarchical inheritance system – that is, you inherit one class from another, creating new classes from existing ones. You can inherit only classes, not regular functions, and variables, in C++. The inherited classes are called *derived* classes, or sometimes, *child* classes. The class that begins the inheritance is called the *base* class or *parent* class. A *single inheritance* is when each derived class is inherited from a single parent class. C++ also supports

```

// Overloading operators illustration involving the List class
#include <iostream>

using namespace std;

class List {
    int a,b,c,d,e; // List values (5Dim Vector)
public:
    List (int num); // constructor
    void printit(void); // prints the list
    List operator+(List & L); // overloads plus
    List operator-(List & L); // overloads minus
    List operator*(List & L); // overloads multiplication
};

List::List(int num) // constructor
{
    a = b = c = d = e = num; // all elements initialized to num
}

void List::printit(void)
{
    cout << "The_result:_[";
    cout << a << ",_" << b << ",_" << c << ",_" << d << ",_" << e << "]" << endl;
}

// all the overloaded operators defined here
List List::operator+(List & arg)
{
    List temp(0); //creates temporary array of zeros
    temp.a = a + arg.a;
    temp.b = b + arg.b;
    temp.c = c + arg.c;
    temp.d = d + arg.d;
    temp.e = e + arg.e;
    return temp;
}

List List::operator-(List & arg)
{
    List temp(0); //creates temporary array of zeros
    temp.a = a - arg.a;
    temp.b = b - arg.b;
    temp.c = c - arg.c;
    temp.d = d - arg.d;
    temp.e = e - arg.e;
    return temp;
}

List List::operator*(List & arg)
{
    List temp(0); //creates temporary array of zeros
    temp.a = a * arg.a;
    temp.b = b * arg.b;
    temp.c = c * arg.c;
    temp.d = d * arg.d;
    temp.e = e * arg.e;
    return temp;
}

int main()
{
    List L1(10), L2(50), L3(0); // initialize three lists

    //check overloaded operators
    L3 = L1 + L2;
    L3.printit();

    L3 = L2 - L1;
    L3.printit();

    L3 = L1 * L2;
    L3.printit();
}

```

Figure 19: list1.cpp: Illustration of math overloading in C++.

```

// auto, static and global variable illustration
#include <iostream>
using namespace std;

void count1(void)
{
    // the following variable - ct - is automatic and local
    int ct1 = 1;

    cout << "count1()_called_" << ct1 << "_time(s)_so_far." << endl;
    ct1++;
}

void count2(void)
{
    static int ct2 = 1; //C++ assigns 1 only the first time
    cout << "count2()_called_" << ct2 << "_time(s)_so_far." << endl;
    ct2++;
}

int ct3 = 1; // global variable declaration

void count3(void)
{
    cout << "count3()_called_" << ct3 << "_time(s)_so_far." << endl;
    ct3++;
}

int main()
{
    cout << "Calling_count1()_three_times." << endl;
    count1();
    count1();
    count1();

    cout << endl << "Calling_count2()_three_times." << endl;
    count2();
    count2();
    count2();

    cout << endl << "Calling_count3()_three_times." << endl;
    count3();
    count3();
    count3();
}

```

Figure 20: static.cpp: Illustration of auto, static and global variables in C++.

multiple inheritance as well. Here a child class can have many parent classes. I am not going to cover multiple inheritance in this lesson.

When one class inherits from another, the derived class inherits all **public** data and member functions from the base class. Therefore, you have to specify only the additional data and functions pertinent to the derived class. Constructors and destructors are never inherited. You must write new constructors and destructors for each derived class when needed. If you want to share **private** members of a class with derived classes, you will have to use the **protected** specifier.

To summarize, **private** members can be used by a classes' own member functions. **public** members can be used by the entire program, as well as by any inherited classes. **protected** members are available to the original member functions and to derived classes, but not to the rest of the program.

These concepts will make sense after we look at a few examples. But staying within this (vague) introductory discussion – even though a derived class have accessible (with appropriate specifications) to **public**, **private** and **protected** members – it is not clear if they end up as **public**, **private** or **protected** members in the derived class. For this we have to understand what is known as *base class access*.

If you do not specify a base class access, C++ assumes that you want **private** inheritance. If the base class access is **private**, *all* the base class's members (both data and functions) will be **private** in the derived class, no matter how they were declared in the base class.

If the base class access is **protected**, the base class's **private** members (data and member functions) remain **private** in the derived class, the base class's **protected** members remain **protected** (inheritable, but still hidden from the rest of the program), and the **public** members change to **protected** members when they are brought to the derived class.

If the base class access is **public**, the base class's **private** members (data and member functions) remain **private**, the base class's **protected** members remain **protected** (inheritable, but still hidden from the rest of the program), and **public** members remain **public** when they are brought to the new derived class.

The code shown below defines a base **Inventory** class, with its own **print()** function – which prints the description of the item in inventory. We have two derived classes – **Auto** and **Machine** – that are derived from the base **Inventory** class. These derived classes must initialize their base class data items to complete the inheritance. The code should be self-explanatory. The resulting output is shown below.

```
The car's data:
Dealer: GM
Quantity: 3
Reorder: 1
Price: $8745.99
Description: Four-Door
```

The machine's data:
 Vendor: Aztec
 Quantity: 11
 Reorder: 5
 Price: \$54.67
 Description: High voltage

```
// Inheritance illustration -- using an inventory program that derives two classes
// called Auto and Machine from base Inventory class
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

//Base class
class Inventory {
    int quant, reorder; // #on-hand & reorder qty
    double price; // price of item
    char * descrip; // description of item
public:
    Inventory(int q, int r, double p, char *); // constructor
    ~Inventory(); // destructor
    void print();
    int get_quant() { return quant; }
    int get_reorder() { return reorder; }
    double get_price() { return price; }
};

Inventory::Inventory(int q, int r, double p, char * d) : quant (q), reorder (r), price (p)
{
    descrip = new char[strlen(d)+1]; // need the +1 for string terminator
    strcpy(descrip, d);
} // Initialization list
Inventory::~Inventory()
{
    delete descrip; // free space & delete description
}
inline void Inventory::print()
{
    cout << "Description:_" << descrip << endl;
}

//Derived Auto Class
class Auto : public Inventory {
    char *dealer;
public:
    Auto(int q, int r, double p, char * d, char *dea); // constructor
    ~Auto(); // destructor
    void print();
    char * get_dealer() { return dealer; }
};

Auto::Auto(int q, int r, double p, char * d, char * dea) : Inventory(q, r, p, d) // base constructor
{
    dealer = new char[strlen(dea)+1]; // need +1 for string terminator
    strcpy(dealer, dea);
}
Auto::~~Auto() { delete dealer; }
void Auto::print() // redefines base class' print()
{
    cout << setiosflags(ios::fixed);
    cout << "Dealer:_" << dealer << endl;
}

//Derived Machine class
class Machine : public Inventory {
    char * vendor;
public:
    Machine(int q, int r, double p, char *d, char * ven); // constructor
    ~Machine();
    void print();
    char * get_vendor() { return vendor; }
};

Machine::Machine(int q, int r, double p, char *d, char * ven) : Inventory(q, r, p, d)//base constructor
{
    vendor = new char[strlen(ven)+1]; // need +1 for string terminator
    strcpy(vendor, ven);
}
Machine::~~Machine() { delete vendor; }
void Machine::print() // redefines base class' print()
{
    cout << setiosflags(ios::fixed);
    cout << "Vendor:_" << vendor << endl;
}
}
```



```

int main()
{
    Auto car(3, 1, 8745.99, "Four-Door", "GM");
    Machine rotor(11, 5, 54.67, "High-voltage", "Aztec");

    // calls to some base class functions to show they are inherited
    cout << setprecision(2); // 2 decimal points
    cout << "The_car's_data:_ " << endl;
    car.print();

    // inherited function calls
    cout << "Quantity:_ " << car.get_quant() << endl;
    cout << "Reorder:_ " << car.get_reorder() << endl;
    cout << "Price:_ $" << car.get_price() << endl;

    //using the base-class's print() function now...
    car.Inventory::print();
    cout << endl << endl;

    cout << "The_machine's_data:_ " << endl;
    rotor.print();

    // inherited function calls
    cout << "Quantity:_ " << rotor.get_quant() << endl;
    cout << "Reorder:_ " << rotor.get_reorder() << endl;
    cout << "Price:_ $" << rotor.get_price() << endl;

    //using the base-class's print() function now...
    rotor.Inventory::print();
    cout << endl << endl;
}

```

6 Input/Output

On many occasions you will have to read (write) data from (to) files. There are many ways to do that. I am going to show some of them using examples. The program shown in figure 21 creates an output file and writes text to it. If the file exists, it is overwritten with the new file. To open and create an output file, you only have to create an output file object and pass that object a filename. The file object in this code is called `outf`, when it goes out of scope, C++ calls the destructor for the output file, which closes the file. The `ifstream` class returns a 0 (null) pointer) if the end-of-file condition is reached. The `while` statement is executed till the end-of-file is reached. The rest of the code should be self-explanatory. In the first method of file input, one character is read at a time, while the other approach reads a 80 character line each time.

The program shown in figure 22 uses the `read()` and `write()` commands to read and write object variables to file. The code should be self-explanatory. A sample run is shown below.

```

* Book Inventory *
What is the title? Sherlock Holmes
Who is the author? A.C. Doyle
What is the price? 10
How many are there? 3

What is the title? Moving from C to C++
Who is the author? G. Perry
What is the price? 5
How many are there? 10

```

```

// A sequential file output and input illustration
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
    cout << "File_creation" << endl;

    // open a file for output by calling the constructor for an output file
    // object
    {
        ofstream outf("MYDATA.txt"); // you could have used a full pathname in the quotes
        outf << "Line_1_in_the_file" << endl;
        outf << "Line_2_in_the_file" << endl;
        outf << "Line_3_in_the_file" << endl;
    } // output object goes out of scope here, and the file gets written on disk.

    char inbuf; // reserve place to hold inout

    cout << "File_input_(one_char_at_a_time)" << endl;

    // open a file for inout by calling the constructor for an input file
    // object
    {
        ifstream inf("MYDATA.txt"); // use pathname if you wish

        while (inf) {
            inf.get(inbuf); // get one character at a time
            cout << inbuf;
        }
    } // output object goes out of scope here

    // alternate approach to read data one line at a time

    char input_buffer[80];

    cout << endl << "File_input_(one_line_at_a_time)" << endl;
    {
        ifstream inf("MYDATA.txt"); // use pathname if you wish

        while (inf) {
            inf.getline(input_buffer, 80); // get one character at a time
            cout << input_buffer << endl;
        }
    } // output object goes out of scope here
}

```

Figure 21: seqout.cpp: Writing and reading from files – an illustration.

Title: Sherlock Holmes

Author: A.C. Doyle

Price: 10 Quantity: 3

Title: Moving from C to C++

Author: G. Perry

Price: 5 Quantity: 10

You can look up other (more efficient) ways of file I/O in C++. For now, this will do.

6.1 Reading Numerical Data from Files

We will have many occasions where we have to read numerical data from files. There are many, many way of getting this done in C++. I am going to provide one method in figure 23. In this example, I expect the first line of a file Data.txt (which is in the current directory) to contain M -many double numbers in the following format

$$\langle P_1 \rangle \quad \langle P_2 \rangle \quad \dots \quad \langle P_M \rangle$$

The C++ code in figure 23 reads these values via the `vector` class. The syntax of its usage (along with a *Google*-search) should tell you everything about this class. The entries in `Data.txt` were

```
4.012  51.32  12.3  345.6
```

The output was

```
Vector P has 4 entries, and they are:
4.012 51.32 12.3 345.6
```

7 Command Line Variables

On many occasions we will need to read some relevant values from the command-line. The C++ code shown in figure 24 should be self-explanatory (after you have seen the code in figure 23). After compilation, when I run the code, I have to provide three command-line inputs. The first is an `int`, the second is a `double`, and the third is a string, which is the name of an input file.

I used the same input file as I did for figure 23. Notice how I run the compiled code, which is shown in figure 25.

References

- [1] G. Perry. *Moving from C to C++*. SAMS publishing, 1992.

```

// read() and write() demonstration
#include <fstream>
#include <iostream>
using namespace std;

class Book { // book inventory class
    char title[50];
    char author[50];
    float price;
    int quantity;

public:
    void ask(); // ask user for data
    void display();
};

void Book::ask() {
    cout << "What is the title?_";
    cin.getline(title, 49);
    cout << "Who is the author?_";
    cin.getline(author, 49);
    cout << "What is the price?_";
    cin >> price;
    cout << "How many are there?_";
    cin >> quantity;
    cin.get(); // this is needed to discard the last <return>
}

void Book::display() {
    cout << endl << "Title:\t" << title << endl;
    cout << "Author:\t" << author << endl;
    cout << "Price:\t" << price << "\tQuantity:_ " << quantity << endl;
}

int main()
{
    Book book1, book2;
    {
        ofstream outf("BOOK.txt");

        cout << " *_Book_Inventory_* " << endl;
        book1.ask();
        outf.write((char *) &book1, sizeof(book1));
        cout << endl;
        book2.ask();
        outf.write((char *) &book2, sizeof(book2));
    }

    const int num = 2;
    Book books[num];
    {
        ifstream inf("BOOK.txt");

        inf.read((char *) &books, sizeof(books));

        for (int ctr = 0; ctr < num; ctr++)
            books[ctr].display();
    }
}

```

Figure 22: read_write.cpp: read() and write() in file I/O – an illustration.

```

// Illustration of how to read numbers from a file
// Written by Prof. Sreenivas for IE523: Financial Computing

#include <iostream>
#include <cmath>
#include <vector>
#include <fstream>

using namespace std;

int main() {
    double value_just_read_from_file;
    vector<double> P; // vector of numbers

    ifstream input_file("Data1.txt"); // The input file name is "Data.txt"
    // It contains the numbers P_1 P_2 ... P_M

    if (input_file.is_open()) // If "Data.txt" exists in the local directory
    {
        while(input_file >> value_just_read_from_file) // As long as you are not at the "end-of-file"
        {
            P.push_back(value_just_read_from_file);
        }
    }
    else
        cout << "Input_file_Data1.txt_does_not_exist_in_PWD" << endl;

    cout << "Vector_P_has_" << P.size() << "_entries,_and_they_are:" << endl;
    for (int i = 0; i < P.size(); i++)
        cout << P[i] << "\t";
    cout << endl;
}

```

Figure 23: numbers_from_file.cpp: Reading numbers from a file – an illustration.

```

// Illustration of how to read values from the Command Line
// Written by Prof. Sreenivas for IE523: Financial Computing

#include <iostream>
#include <cmath>
#include <vector>
#include <fstream>

using namespace std;

int main (int argc, char* argv[])
{
    // First command line variable is an integer
    int first_command_line_variable;
    sscanf (argv[1], "%d", &first_command_line_variable);

    // Second command line variable is a double
    double second_command_line_variable;
    sscanf (argv[2], "%lf", &second_command_line_variable);

    // Third command line variable is an input Filename, that I use
    // later
    ifstream input_file (argv[3]);

    double value_just_read_from_file;
    vector <double> P; //vector of numbers

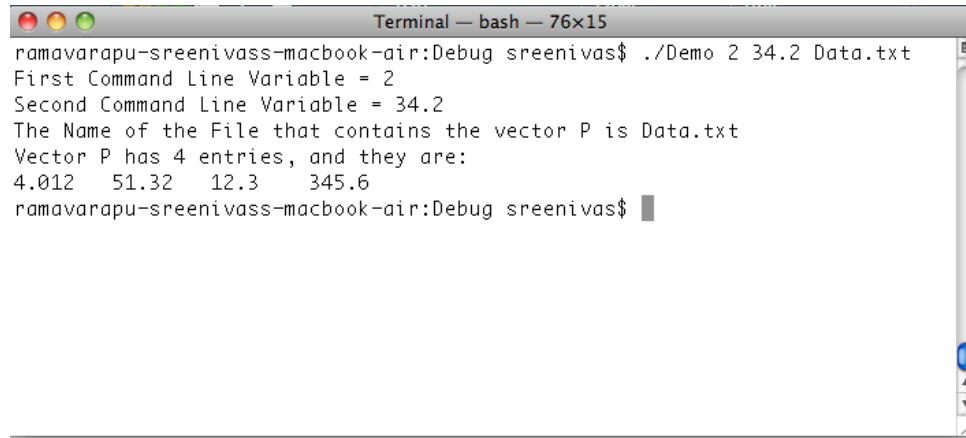
    // The input file name is "Data.txt"
    // It contains the numbers P_1 P_2 ... P_M

    if (input_file.is_open()) // If "Data.txt" exists in the local directory
    {
        while(input_file >> value_just_read_from_file) // As long as you are not at the "end-of-file"
        {
            P.push_back(value_just_read_from_file);
        }
    }
    else
        cout << "Input_file_Data.txt_does_not_exist_in_PWD" << endl;

    cout << "First_Command_Line_Variable_=" << first_command_line_variable << endl;
    cout << "Second_Command_Line_Variable_=" << second_command_line_variable << endl;
    cout << "The_Name_of_the_File_that_contains_the_vector_P_is_" << argv[3] << endl;
    cout << "Vector_P_has_" << P.size() << "entries, and they are:" << endl;
    for (int i = 0; i < P.size(); i++)
        cout << P[i] << "\t";
    cout << endl;
}

```

Figure 24: command_line.cpp: Taking input values from the command line – an illustration.



```
Terminal — bash — 76x15
ramavarapu-sreenivass-macbook-air:Debug sreenivas$ ./Demo 2 34.2 Data.txt
First Command Line Variable = 2
Second Command Line Variable = 34.2
The Name of the File that contains the vector P is Data.txt
Vector P has 4 entries, and they are:
4.012  51.32  12.3  345.6
ramavarapu-sreenivass-macbook-air:Debug sreenivas$
```

Figure 25: Illustration of the command line inputs for the C++ code shown in figure 24.