

# Day 3

## Functions: Pascal to C++

### Functions

Pascal has procedures and functions. C++ has functions. A few very simple examples will demonstrate the similarities between Pascal and C++ . Let's start with a Pascal function which accepts two integers and returns the larger of the two.

Function Max (a,b : integer): integer;

Var

temp : integer; (\* This variable is unnecessary and here only to demonstrate local variables\*)

Begin

if a > b then

temp := a

else

temp := b;

max := temp;

End;

### The corresponding C++ function is

```
int max (int a, int b)
```

```
{
```

```
int temp;
```

```
if (a > b)
```

```
temp = a;
```

```
else
```

```
temp = b;
```

```
return temp;
```

```
}
```

In both Pascal and C++, functions may be used in ordinary expressions.

In Pascal:

```
writeln('Enter two numbers ');
```

```
readln(x,y);
```

```
writeln('The square of the larger number is ', max(x,y)*max(x,y));
```

In C++ :

```
cout<<"Enter two numbers "<<endl;
```

```
cin>>a>>b;
```

```
cout<<"The square of the larger number is "<< max(x,y)*max(x,y)<<endl;
```

There are just a few trivial differences between the Pascal and C++ versions of the max function.

1. In the heading, the return value, (int in the C++ version) comes before the name of the function. (In Pascal, it comes at the end.)
2. In Pascal, the return value is assigned to the function name ( max := temp); in C++ we use a *return* statement which causes the function to return program control to the caller. The syntax is *return value*; where *value* may be an expression. A function may not return an array. Remember an array is a **constant** pointer. (We'll see more on this later. Any other data type (including a structure) may be returned.

In our examples, the parameters *x* and *y* are both passed to the functions **by value**. Thus when function *max* is called, with parameters *x* and *y*, the **values** stored in *x* and *y* are copied into (local) variables *a* and *b*. The consequence of this is that *x* and *y* are *protected* from the actions of the function. Any actions affecting *a* or *b* will not affect *x* or *y*. When the function returns, *x* and *y* are unchanged.

The variable *temp* in both versions is a local variable. This variable, *temp*, is created when the function is called and destroyed when the function exits. Variable *temp* is known only within the function.

In addition to functions, Pascal also supplies *procedures*. The following very standard Pascal procedure swaps the values stored in two variables.

```

Procedure swap (var a, b : integer);
Var
    temp : integer;

Begin
    temp := a;
    a := b;
    b := temp
End;
```

As we have already said, C++ supports functions and only functions. Consequently, we will write a C++ *function* which swaps the values stored in two variables. To indicate that this function will not return a value, we use the return type *void*. (This is the idea behind using the word *void* in our main function.) Thus the C++ version is as follows:

```

void swap (int & a, int & b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
    return; // this empty return is really optional
}
```

Notice that in the heading of the Pascal version the keyword *var* is included. As you know, *var* indicates that the parameters are passed **by reference**. Consider the following code segment:

```
x, y : integer;
Procedure swap (var a, b : integer);
Var
    temp : integer;
```

**Begin**

```
    temp := a;
    a := b;
    b := temp
```

End;

Begin

```
    x := 3;
    y := 5;
    write(x,y);
    swap(x,y);
    writeln( x,y)
```

End;

The output is 3 5 5 3.

When function *swap* is invoked, the values stored in *x* and *y* are **not** copied to *a* and *b*. In this case, *a* and *b* point to *x* and *y*. Parameters *a* and *b* hold the addresses of *x* and *y*. If *a* and *b* are altered so are *x* and *y*.

In C++, we accomplish this by using references (`void swap (int &a, int &b)`). Here *a* and *b* are references or aliases for the actual parameters which are passed to the function. The effect is the same as in the Pascal procedure: the procedure changes the values of the actual parameters which are passed to *a* and *b* and *a* and *b* hold memory addresses.

In C (not C++) all parameters are passed by value. References are not supported in C. So how do we write a simple swap procedure in C? We **must** use pointers.

```
void swap ( int * a, int * b) // notice the pointer notation
{
    int temp;
    temp = *a; // contents of a
    *a = * b;
    *b = temp;
    return;
}
```

void main ()

```
(
    int x = 3;
    int y = 5;
    swap (&x, &y); // pass the ADDRESS of x and y
```

```
}
```

In the above segment, the parameters are passed **by address**. This of course will work in C++; however by introducing references, C++ has eliminated all the cumbersome pointer notation of C. Behind the scenes, things really work the same whether we are passing by reference or by address. C++ has eliminated all the pointer notation.

### ***Arrays and Functions.***

To pass an array to a C++ function use the syntax as in the following function heading:

```
void name ( int a[], int size)
```

The name *a* denotes an array of integers, and *size* is an integer which indicates the size of the array. This parameter (*size*) is not required. The brackets *[]* indicate that *a* is an array. We do not include the size of the array in the brackets. (When we pass a two dimensional array, however, we must include the second dimension : *void name (int a[][5], int rows, int columns)* ).

Recall that the name of an array is a pointer to the beginning of the array i.e. if *a* is the name of an array then *a* is the address of *a[0]*. Further, we have seen that *a[i]* is simply notation for *\*(a+i)*. Thus, when passing an array, we are really passing (by value) a **pointer**, the address of the first element in the array—not an entire array. Consequently, the entire array is not copied. We can access elements of the array by using standard array notation (*a[i]*) or pointer notation (*\*(a+i)*). In either case, changes made in the function will affect the array since **no copy of the entire array is created**.

The following example implements a previous bubblesort program as a collection simple functions. The program also illustrates a few new concepts.

```
#include <iostream.h>
const int max = 200; // maximum size of array
struct student
{
    int id;
    float gpa;
};

student list[max]; //index range : 0 to 199
int n = 0; // counts number of data

void readdata (student list[], int & n) // notice parameters
{
    student data;
    n = 0;
    cout<<"Enter data—one student per line—end with CTRL-Z"<<endl;
    while ( cin>>data.id>>data.gpa)
    {
        n++;
        list[n-1]=data; // arrays indexed from 0
    }
}
```

```

    }
}

void swap (student& a, student& b)
{
    student temp = a;
    a =b;
    b = temp;
}

void bubblesort (student list[], int n)
{
    bool interchange = true; //Do we make a swap? Yes or no?
    int pass = 1;           // Counts the number of passes through the array

    while ((interchange ) && (pass <= n-1)) // array has n elements ; at most n-1 passes
    {
        interchange = false;
        for (int i = 0; i <= (n-1)-pass; i++) //i is known only in the enclosing (while) block
            if (list[i].gpa < list[i+1].gpa)
            {
                interchange = true;
                swap(list[i],list[i+1]);
            }
        pass++;
    }
}

void outputdata (student list[], int n)
{
    cout<<"GPA and Student ID—sorted by GPA "<<endl;
    for (int i = 0; i <= n-1; i++) // note: i must once again be declared.
        cout<<list[i].gpa<< " "<<list[i].id<<endl;
}

void main ()
{
    readdata(list,n);
    bubblesort(list, n);
    outputdata(list,n);
}

```

In the preceding program, you no doubt noticed the variables declared at the top of the program which are not included in any function. These are **global** variables. Like Pascal, C++ has both global and local variables. Variables declared within a function are local variables and those

declared outside a function are global. The *scope* of a variable is that section of the program where the variable is known. The scope of a global variable extends from its declaration to the end of the file in which it is declared. The scope of a local variable is the block in which it is declared. Thus if a local variable is declared at the top of a function, it is known throughout that function and only that function. If a local variable is declared within a block in a function, it is known only within that block:

```
void afunction (int x)
{
    int a; // known throughout the entire function
    while (somecondition)
    {
        int b ; // known only in the enclosing block
    }
    // a is known here but b is not know here
}
```

Returning to the program, we see three functions. All three have *void* return types. The first function, *readdata*, takes two parameters: an array and an integer. Notice that the second parameter is passed by reference. This function will fill the array and return its size via parameter *n*. Function *bubblesort* will sort the array, so *n* is passed by value and procedure *outputdata* will print the array, so again *n* is passed by value and *n* will not be changed by these functions.

Notice that the *swap* function was not nested within the *bubblesort* function. **C++ does not allow nested functions.** Pascal and C++ differ in this respect.

### ***Using the const modifier with parameters.***

In our example, the array *list* was passed to three functions. The functions *readdata* and *bubblesort* both altered the array ( one filled the array the other sorted the array). The function *outputdata* did not change the array, yet function *outputdata* had the capability of doing so. The question arises: “How do we pass an array to a function and still protect the array?” This is easily done with the *const* modifier. In the function *outputdata* we would simply change the declaration to

```
void outputdata ( const student list [], int n)
```

Now if the function tried to alter the array, the compiler would issue an error message.

The *const* modifier is also an alternative to passing structures by value. Recall that when we pass a parameter by value, a **copy** of the parameter is made. A struct may be quite large and making a copy of a struct could be quite inefficient. To circumvent copying the entire struct and still protect the struct we can again use the *const* modifier: we pass the struct by reference along with the keyword *const*.

```
struct student
{
```

```

    int id;
    float gpa;
};

void dosomething (const student& s)

```

Here, we are passing the structure by reference but the *const* modifier prevents the procedure from altering its contents. In this case, we have passed the parameter by *constant reference*.

### The static modifier.

A local variable declared in a function, is destroyed when the function exits. Thus the value stored in a local variable is lost. By using the *static* modifier, a variable declared in a function will not be destroyed and will retain its value throughout the life of the program. In the following example assume we have several functions which play games like blackjack or tic-tac-toe. We will use a local variable to keep track of the number of wins for each game. Each game will have its own local *numwon*.

```

#include <iostream.h>

void tictactoe()
{
    static int numwon = 0; // static variable will retain its value beyond the life of the
                          //function
    static int gamenumber = 0;
    gamenumber++;

    // code to play game goes here

    if (won)
        numwon++;
    cout <<"Total tic-tac-toe won " << numone<<" out of " << gamenumber <<endl;
}

void blackjack() // notice there are no parameters
{
    static int numwon = 0; // static variable will retain its value beyond the life of the
                          //function

    static int gamenumber = 0;
    gamenumber ++;
    // code to play game goes here

```

```

    if (won)
        numwon++;
    cout <<"Total blackjack won " << numwon<<" out of " << gamenumber <<endl;
}

```

//Other games (craps, poker....) could go here.

```

void main( )
{
    // code which presents a menu of games then runs the selected game
}

```

In the above program shell, the local variable, *numwon* and *gamenumber* are initialized **once** to 0. When the procedure exits, *numon* and *gamenumber* retain their current values. Notice the modifier *static* in the declaration. Variables which are destroyed when a function exits are called *automatic*.

## Function Overloading

In the bubblesort example, we included a swap function which interchanges two structs of type student. In a large program we might need several different swap functions: one which swaps integers, another which swaps floats, etc. We could name each of these appropriately (swapstudent, swapint, swapfloat); however this proliferation of names is somewhat awkward. Instead C++ allows us to *overload* a function name so that we can use the same name for several different functions. For example, we might have three different swap functions:

```

void swap (int& a, int& b)    void swap( student& a, student& b)    void swap( float& a, float& b)
{
    int temp = a;            {
        student temp = a;            {
            a = b;                    float temp = a;
            b = temp;                a = b;
        }                            b = temp;
    }                                }

```

When we invoke function *swap*, the compiler decides on the correct version based on the parameters we pass. To overload a function name the different versions of the function must differ in the number and/or type of parameters. In other words, the compiler must be able to choose the correct version of the function from examining the argument list.

Let's look at another example.

```

struct triangle
{
    float base;
    float alt;
};
struct rectangle

```

```

{
    float length;
    float width;
};

struct trapezoid
{
    float alt;
    float base1;
    float base2;
};

float area (triangle x)
{
    return .5*x.base*x.alt;
}

float area (rectangle x)
{
    return x.length*x.width;
}

float area (trapezoid x)
{
    return .5*x.alt*(x.base1+x.base2);
}

```

Here we have three different functions called *area*. Each accepts a different type of argument and each performs a different task. The name *area* is overloaded.

There is a major difference in the two examples of overloaded functions above. The "swap group" used the same algorithm (method of computation) but the "area functions" each used a different algorithm. It seems somewhat redundant to write three different swap functions with essentially the same code. It should be no surprise that, C++ supplies a remedy for this redundancy of code.

### **Function Templates.**

In C++, we may write one generic swap function which will accept data of any type. Such a generic function is called a *function template*. Here's how it's done:

```

template < class T>
void swap (T& a, T& b)
{
    T temp = a;
    a =b;
    b = temp;
}

```

```
}
```

The keyword *template* is used to indicate we are defining a generic or template function. The words *class T* indicate that *T* will be a placeholder for some type. (There is nothing special about *T* we could have used *<class Dingbat>*.) Now throughout the definition of the function, we use the name *T* in place of a type name. We may now call the function passing it two ints, two structs, two floats or whatever makes sense.

For example, suppose we have several different types of structures:

```
struct student
{
    int id;
    char name[25];
    float gpa;
};
```

```
struct faculty
{
    int id;
    char name[25];
    float salary;
};
```

```
struct staff
{
    int id;
    char name[25];
    float wage;
    float hours;
};
```

We can write one sort routine which will sort an array of student, an array of faculty, or an array of staff by id:

```
template < class T> // T is a placeholder for any type
void swap < T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
}
```

```
template <class type> // type is a placeholder for any type
void sort (type list[], int n)
{
```

```

bool interchange = true;
int pass = 1;

while ((interchange ) && (pass <= n-1))    // array has n
{
    interchange = false;
    for (int i = 0; i <= (n-1)-pass; i++)
        if (list[i].id < list[i+1].id)    // note struct must have id field
            {
                interchange = true;
                swap(list[i], list[i+1]);
            }
        pass++;
    }
}

```

This one sort routine will accept an array of any struct with a numeric *id* field. Pascal would require a different function for each separate type. We could just as easily write a single function template which sorts an array of any simple type (int, float, double, char). Again, Pascal would require a separate routine for each type.

### Function Prototypes - Declaration vs. Definition

Pascal demands that the layout of your program conforms to a certain standard: constant section, type section, variable section, functions and procedures and finally the main body. C++ is not so picky. You have already seen that in C++ variables may be declared at any point in a program. Likewise, functions need not be defined in any particular region in a program. The only restriction is that a function must be *declared* before it is used. A function in C++ may be *declared* using a *function prototype*. A *function prototype* consists of the function name, the return type and the argument types and ends with a semicolon-- that's all. Here is a short example illustrating the use of function prototypes where the body of the function is defined elsewhere.

```

#include <iostream.h>

// global variables
const int max = 200; // maximum size of array
struct student
{
    int id;
    float gpa;
};

student list[max]; //index range : 0 to 199
student data;
int n = 0; //number of elements stored in list

```

```
// these are function prototypes---function DECLARATIONS
// C++ requires the name, return type and argument types before a function is used
// The actual definition of a function may come elsewhere
```

```
void readdata (student [], int & ); // note the array syntax
    // reads n data into an array of type student
```

```
void swap (student& , student& );
    // swaps two structures
```

```
void bubblesort (student [], int);
    // sorts an array of type student on the gpa
```

```
void outputdata (const student [], int);
    // prints an array of type student ( id, gpa)
```

```
void main ()
{
    readdata(list,n);
    bubblesort(list, n);
    outputdata(list,n);
}
```

```
// These are the function DEFINITIONS
```

```
void readdata (student list[], int & n)
{
    cout<<"Enter data—one student per line—end with CTRL-Z"<<endl;
    while ( cin>>data.id>>data.gpa)
    {
        n++; // note "final" array contains n+1 elements
        list[n-1]=data;
    }
}
```

```
void swap (student& a, student& b)
{
    student temp = a;
    a =b;
    b = temp;
}
```

```
void bubblesort (student list[], int n)
{
    bool interchange = true; //Do we make a swap? Yes or no?
```

```

int pass = 1;          // Counts the number of passes through the array

while ((interchange ) && (pass <= n-1))
{
    interchange = false;
    for (int i = 0; i <= (n-1)-pass; i++) //i is known only in the enclosing (while) block
        if (list[i].gpa < list[i+1].gpa)
            {
                interchange = true;
                swap(list[i], list[i+1]);
            }
    pass++;
}

void outputdata (const student list[], int n)
{
    cout<<"GPA and Student ID—sorted by GPA "<<endl;
    for (int i = 0; i <= n-1; i++) // note: i must once again be declared.
        cout<<list[i].gpa<< " "<<list[i].id<<endl;
}

```

Notice that the prototypes did not include parameter names, only types. Parameter names may be included in a prototype, but they are not required. Thus the above prototypes may have been written as

```

void readdata (student list[], int & n); // note the array syntax
void swap (student& a, student& b );
void bubblesort (student [] list , int n);
void outputdata (const student list [], int n);

```

In fact, the parameter **names** in a prototype need not match the names in the function definition. Parameter names in the prototypes are ignored by the compiler. Separating function declarations from the corresponding definitions is of course optional. Nonetheless, in a large program, with many functions, such a separation can add to the overall clarity and readability of the program.

Another common practice is placing declarations in a *header file* and definitions in an *implementation file*. If we were to put the declarations for the previous example into a header file, say *sortstudent.h* and the definitions in *sortstudent.cpp* the structure of our program would take the form:

```

#include <iostream.h>
#include "sortstudent.h"
#include "sortstudent.cpp"
void main ()

```

```

{
    readdata(list,n);
    bubblesort(list, n);
    outputdata(list,n);
}

```

Again, for small programs, such as this one, there is really nothing to be gained by this separation.

### C++ library functions.

We have already seen the library `string.h` and the `iostream.h` library. C++ supplies a host of other libraries. The library `math.h` contains all the standard mathematical functions like `sin`, `cos`, `tan`, `sqrt`, `exp`, `log`, etc. The library `stdlib.h` contains the random number generator `random(n)`, which produces a random integer in the range 0 to n-1. The following program demonstrates the random number generator supplied by `stdlib.h`. The program will roll two dice until a 7 or 11 is rolled (win) or a 2,3, or 12 is rolled (lose).

```
#include <iostream.h>
```

```
#include <stdlib.h> // for random number generator
```

```

int rolldice (); // (prototype) returns sum of two dice -- an integer in the range 2..12
void main()
{
    int dice;
    randomize(); // seeds the random number generator using time
                 // without a call to randomize, each run of the program
                 // would produce the same sequence of "random" numbers
    do
    {
        dice = rolldice();
        cout<<dice<<endl;
    }
    while ((dice !=2) && (dice !=3) && (dice !=7) && (dice != 11) && (dice != 12));
    if ((dice ==7) || (dice == 11))
        cout <<"WIN"<<endl;
    else
        cout <<"LOSE"<<endl;
}

```

```
int rolldice () // function definition
```

```

{
    // adds two random numbers in the range 1 to 6
    int die1;
    int die2;
    die1 = random(6) + 1; // random(n) returns an int between 0 and n-1 inclusive
    die2 = random(6) + 1; // so random(6) returns one of 0,1,2,3,4 or 5
    return die1+die2;
}

```



## Exercises for Day 3

### Shorter Exercises

#### 1. Average with Functions (\*)

Write a function which accepts three parameters of type `double` and returns the average of these numbers. Test your function using a main function.

#### 2. Arithmetic with Functions

Write a function with three parameters (`double`). The function adds 10% to each parameter, and returns the average of the three new values. The actual parameters should be changed after the function call. Test your function by reading three values in main, calling your function and printing the three old values, three new values, the average of the original three values and the average of the three new values..

#### 3. Functions with Array Parameters (\*)

Write a function with two parameters, an array of positive integers and the size of the array. The function should return the average (type `double`) of the values stored in the array. Also, include a main function which (1) reads a list of (positive) integers into an array until a negative number is encountered (2) passes the array and its size to your (average) function, and (3) prints the resulting average.

#### 4. Functions with String Parameters (\*)

Write a function which takes any three strings as input, and returns a string formed by concatenating the three strings in alphabetical order. Test your function with a main function.

#### 5. A Mathematical Function – The Least Common Multiple

Write a function with three integer parameters which returns the smallest integer that is a multiple of all three integers. For example, if the values passed to the function are 2,3,and 4, your function should return 12. Test your function using a main function. Assume the integers are positive.

## 6. Golf in the Arrays

Write a function which accepts two integer arrays, each of size 18. The first array represents par score on each of 18 holes in a golf course. The second array represents the actual scores of a player's round. Your function should return an integer that represents how many strokes the player's round was above or below par. (Your function should return 0 if the player shot even par). Test your function with a main function.

## Longer Exercises

### 7. MAXSORT Using Functions (\*)

Let's rewrite MAXSORT using functions.

- a. Write a function called MAX which accepts an array **A** of strings and returns the index of the alphabetically largest string among the first *n* strings. The array **A** and *n* should be parameters. Also, MAX should use the **strcmp** function
- b. Write a function MAXSORT to sort an array **A** of **numStrings** strings. On each iteration, MAXSORT uses the MAX function to find the index of the maximum element in that part of **A** indexed from 0 to **last** and swaps this maximum element with the element in position **last**. The value of **last** is initially set to **numStrings** - 1 and is decremented at each iteration until it is 0. In other words, on each iteration, MAXSORT places the next largest element into its proper position. MAXSORT should use a SWAP function. The SWAP function must use **strcpy** and hence cannot use the function template in this chapter.
- c. Write a function that prints out an array **A** of **size** strings.
- d. Write a function that reads a sequence of strings into an array **A** and returns the number of strings which were read.
- e. Write a main function that creates a dynamic array of dynamic strings. The user should be asked for the number of strings to be read. Your program should print the sorted array.

## 8. Games of Chance and Simulations (\*)

Sometimes the easiest way for a casino to analyze whether or not a particular game of chance is favorable is to simulate the game by computer and analyze the results. The analytical alternative of rigorous mathematical analysis is not always easy or practical.

The game of CRAPS is played by rolling two dice repeatedly until the player wins or loses. If the player rolls a 7 or 11, she wins immediately and the game is over. If she rolls a 2, 3 or 12, she loses immediately and the game is over. Otherwise, her roll is remembered and this value is called her point. This point stays fixed for the rest of the game. If she subsequently rolls her point again before she rolls a 7, she wins. If she rolls a 7 before she rolls her point again, she loses. The number 7 is called the “point breaker”. Once a point is made, the numbers 2, 3, 11 and 12 are irrelevant. Only the point and the point breaker are considered.

- a. Write a function that plays a variation of CRAPS, and returns **true** if the player wins or **false** if the player loses. In this version, numbers for winning and losing on the first roll of the dice are set by the user, but the other rules of the game remain the same. The point breaker should be the smallest of the (first roll) winning numbers (e.g. 7 in the normal game since 7 and 11 are immediate winners). The input is a sequence of characters which is stored in an array, A, of size 11. Each character can be W, L, or N -- W for win, L for lose, or N for neither. The character stored in A[i] (i ranges from 0 to 10) indicates whether the value i + 2 is an immediate (first roll) win, an immediate loss or neither. For example, in the standard game of craps 7 and 11 are immediate wins, 2, 3, and 12 are immediate losses and 4,5,6,8,9, and 10 are neither immediate wins nor losses. Thus the array A contains the following character values:

```
A[0] = 'L' (since 2 is an immediate loss)
A[1] = 'L' (since 3 is an immediate loss)
A[2] = 'N' (4 is neither win nor loss)
A[3] = 'N'
A[4] = 'N'
A[5] = 'W' (7 is an immediate win)
A[6] = 'N'
A[7] = 'N'
A[8] = 'N'
A[9] = 'W' (11 is an immediate win)
A[10] = 'L' (12 is an immediate loss)
```

Another variation might have A = { 'W', 'L', 'N', 'N', 'N', 'L', 'L', 'W', 'N', 'W', 'L' }  
For this game a 2,9 or 11 wins on the first roll; 3,7,8, or 12 loses on the first roll and 4,5,6 or 10 neither win nor lose on the first roll. The point breaker is 2, since 2 is the lowest of the winning numbers 2.

b. Write a main function which reads a sequence of 11 characters (W,L, or N) , plays 100 games of this version of craps and reports the results. Try your function on the standard variation and on the variation where 3 is no longer an immediate loss.

## 9. Functions with Pointer Parameters

a. In a file called `linked.h`, write *declarations* for the following functions and data structures. In a file called `linked.cpp`, write the *definitions* for the following functions and data structures.

- i. A linked list structure of **student**. The data structure **student** should have 4 fields: a string of max length 25 for the name, an integer id, a float gpa, and a character (m/f) for gender.
  - ii. A function to read a sequence of student data and construct a linked list.
  - iii. A function to print a list of student data
  - iv. A function that takes a list and returns the number of nodes in it.
  - v. A function that calculates the average gpa of all the persons in a list.
  - vi. A function that takes two lists and returns **true** when they are equal and **false** otherwise. For our purposes, we consider two lists to be “equal” when they contain the same names in the same order. (What are other options?)
- b. Write a main function which reads in a sequence of student data, builds a linked list, prints the number of students on the list, prints the average gpa of all the students on the list, and prints the names of the students in reverse order from that in which they were input. Include `linked.h` in a file with your main function. Compile this file. You will be able to compile this file but not run it. (Why not?)
- c. Compile `Linked.cpp`,
- d. Include `linked.h`, `linked.cpp` and your main function in one file. Run your complete program. (If you know how to separately compile and link these files, you might try that!)

### Note on Interface Versus Implementation:

This problem emphasizes the distinction of the declaration of functions and data structures from the definition of functions and data structures. The idea is to separate the interface from the implementation. The problem foreshadows the notion of classes in the next chapter where the interface and implementation of an object are clearly distinguished.

