C/C++ Program Design

LAB 9

CONTENTS

- Learn the concept of storage duration, scope and linkage
- Learn to use namespaces
- Learn to create and use static library and dynamic library

2 Knowledge Points

2.1 Storage duration, Scope and Linkage

2.2 Namespaces

2.3 Static library and Dynamic library

2.1 Storage duration, Scope and Linkage

- **Scope** describes the region or regions of a program that can access an identifier. An identifier has one of following scopes: **block scope**, **function prototype scope**, or **file scope**.

- **Linkage** describes how a name can be shared in different units. A variable has one of the following linkages: **external linkage**, **internal linkage**, or **no linkage**. A name with external linkage can be shared across files, and a name with internal linkage can be shared by functions within a single file. Names of automatic variables have no linkage because they are not shared.

- **Storage duration** describes the lifetime of a variable. A variable has one of the following storage durations: automatic storage duration, static storage duration, dynamic storage duration or thread storage duration.

C++ uses three separate schemes(four under C++11) for storing data. The different storage classes offer different combinations of scope, linkage and storage duration.

- **Automatic storage duration**: **Variables declared inside a function definition**(including function parameters)have automatic storage duration. They are created when program execution enters the function or block in which they are defined, and the memory used for them is freed when execution leaves the function or block.
- **Static storage duration**: **Variables defined outside a function definition** or else by using the **keyword static** have static storage duration. They persist for the entire time a program is running.
- **Dynamic storage duration**: Memory allocated by the **new operator** persists until it is freed with the delete operator or until the program ends, whichever comes first. This memory has dynamic storage duration and sometimes is termed the free store or the heap.
- **Thread storage duration(C++11)**: Variables declared with the **thread_local** keyword have storage that persists for as long as the containing thread lasts.

2.1.1 Automatic Storage Duration

Function parameters and variables declared inside a function have, by default, automatic storage duration. **They also have local scope and no linkage**.

autoduration.cpp > ...

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

int main()
{
    int x = 30;   // original x

    cout << "x in outer block: " << x << " at " << &x << endl;
    {
        int x = 77;   // new x, hide the original x
        cout << "x in inner block: " << x << " at " << &x << endl;
    }
    cout << "x in outer block: " << x << " at " << &x << endl;

    while(x++ < 33)   // original x
    {
        int x = 100;  // new x, hide the original x
        x++;
        cout << "x in while loop: " << x << " at " << &x << endl;
    }
    cout << "x in outer block: " << x << " at " << &x << endl;

    return 0;
}
```

```
x in outer block: 30 at 0x7ffe1da7d240
x in inner block: 77 at 0x7ffe1da7d244
x in outer block: 30 at 0x7ffe1da7d240
x in while loop: 101 at 0x7ffe1da7d244
x in while loop: 101 at 0x7ffe1da7d244
x in while loop: 101 at 0x7ffe1da7d244
x in outer block: 34 at 0x7ffe1da7d240
```

2.1.2 Static Storage Duration

C++, like C, provides **static storage duration variables** with three kinds of linkage: **external linkage** (accessible across files), **internal linkage** (accessible to functions within a single file),and **no linkage** (accessible to just one function or to one block within a function).

All three last for the duration of the program. The static variables stay present as long as the program executes.

1.Static Duration, External Linkage

Variables with **external linkage** are often simply called external variables(global variables). They necessarily have static storage duration and file scope. External variables are defined outside, and hence external to any function.

If you use an external variable in several files, only one file can contain a definition for that variable (per the one definition rule). But every other file using the variable needs to declare that variable using the keyword extern.

```
// file01.cpp
extern int cats = 20;     // definition because of initializations
int dogs = 22;           // also a definition
int birds;              // also a definition
...
```

```
// file98.cpp
// use cats, dogs, and birds from file01.cpp
extern int cats;
extern int dogs;
extern int birds;
...
```

```
int  p=1,q=5;
extern  char c1,c2;
float  f1(int a)
{  int b,c;
   extern  char c1,c2;
     .......
}
int   f3()
{.....
}
char c1,c2;
char f2(int x,int y)
{   int i,j;
     ......
}
void main()
{  int m,n;
    .......
}
```

extern c1,c2 scope

extern c1,c2 scope

p,q scope

c1,c2 scope

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

int x;
int  main()
{
    int x = 256;

    cout << "local variable x = " << x << endl;

    cout << "global variable x = " << ::x << endl;

    return 0;
}
```

Declare a global variable whose initial value is 0

Declare a local variable whose name is the same as the global variable.

The local variable hides the global variable.

Using **scope-solution operator(::)** to access the global variable.

```
local variable x = 256
global variable x = 0
```

2. Static Duration, Internal Linkage

Variables of this storage class have static storage duration, file scope, and internal linkage. You can create one by defining it outside of any function (just as with an external variables) with the storage class specifier **static**. A variable with internal linkage is local to the file that contains it.

```
// file1
int errors = 20;        // external declaration
...
------------------------------------------
// file2
int errors = 5;         // ??known to file2 only??
void froobish()
{
      cout << errors;   // fails
      ...
```

external variable

```
// file1
int errors = 20;        // external declaration
...
------------------------------------------
// file2
static int errors = 5;  // known to file2 only
void froobish()
{
      cout << errors;   // uses errors defined in file2
      ...
```

Using **static** to share data among functions found in just one file, avoiding name conflicting with external variable.

3. Static Duration, No Linkage

You create such a variable by applying the **static** modifier to a variable defined **inside a block**. When you use it inside a block, static causes a local variable to have static storage duration. If you initialize a static local variable, the program **initializes the variable once**.

staticduration.cpp > main()

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

void trystat();

int main()
{
    for(int count = 1; count <= 3; count++)
    {
        cout << "Here comes iteration " << count << ":\n";
        trystat();
    }

    return 0;
}

void trystat()
{
    int fade = 1;
    static int stay = 1;

    cout << "fade = " << fade++ << " and stay = " << stay++ << endl;
}
```

auto variable

Static variable

```
Here comes iteration 1:
fade = 1 and stay = 1
Here comes iteration 2:
fade = 1 and stay = 2
Here comes iteration 3:
fade = 1 and stay = 3
```

factorial.cpp > factorial(int)

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

long factorial(int n);

int main()
{
    for(int i = 1; i <= 5; i++)
        cout << i << "!= " << factorial(i) << endl;

    return 0;
}

long factorial(int n)
{
    static long product = 1;

    product *= n;

    return product;
}
```

```
1!= 1
2!= 2
3!= 6
4!= 24
5!= 120
```

C and C++ use scope, linkage, and storage duration to define five storage classes: automatic, register, static with block scope, static with external linkage, and static with internal linkage.

Five Storage Classes

Storage Class	Duration	Scope	Linkage	How Declared
`automatic`	Automatic	Block	None	In a block
`register`	Automatic	Block	None	In a block with the keyword `register`
`static` with external linkage	Static	File	External	Outside of all functions
`static` with internal linkage	Static	File	Internal	Outside of all functions with the keyword `static`
`static` with no linkage	Static	Block	None	In a block with the keyword `static`

partb.cpp > ...

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

extern int count;    //reference declaration, external linkage

static int total = 0; //static definition, internal linkage

void accumulate(int n)    //n has block scope, no linkage
{
    static int subtotal = 0; // static, no linkage

    if(n <= 0)
    {
        cout << "loop cycle: " << count << endl;
        cout << "subtotal: " << subtotal << ", total: " << total << endl;
        subtotal = 0;
    }
    else
    {
        subtotal += n;
        total += n;
    }
}
```

Static variable

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples$ g++ parta.cpp partb.cpp
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples$ ./a.out
Enter a positive integer(0 to quit):5
loop cycle: 1
subtotal: 15, total: 15
Enter a positive integer(0 to quit):10
loop cycle: 2
subtotal: 55, total: 70
Enter a positive integer(0 to quit):2
loop cycle: 3
subtotal: 3, total: 73
Enter a positive integer(0 to quit):0
Loop executed 3 times.
```

parta.cpp > ...

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

void report_count();
void accumulate(int n);
int count = 0;    //file scope, external linkage

int main()
{
    int value;    //automatic variable
    register int i;   //register variable

    cout << "Enter a positive integer(0 to quit):";
    while(cin >> value)
    {
        if(value == 0)
            break;
        if(value > 0)
        {
            ++count;
            for(i = value; i >= 0; i--)
                accumulate(i);
        }
        cout << "Enter a positive integer(0 to quit):";
    }
    report_count();

    return 0;
}

void report_count()
{
    cout << "Loop executed " << count << " times.\n";
}
```

Calling the function for several times

2.2 Namespace

Namespaces provide a much more controlled mechanism for preventing name collisions. A namespace is a scope.

Namespace definition

keyword namespace name

```
namespace nsp{
    // variables (with their initializations)
    // structure declaration
    // functions (with their definitions)
    // templates declaration
    // classes declaration
    // other namespaces
}
```

There is no semicolon.

```cpp
namespace Jack{
    double pail;                    // variable declaration
    void fetch();                   // function prototype
    int pal;                        // variable declaration
    struct Well{ ... };             // structure declaration
}

namespace Jill{
    double bucket(double n){ ... }  // function definition
    double fetch;                   // variable declaration
    int pal;                        // variable declaration
    struct Hill{ ... };             // structure declaration
}
```

This two variables are not conflict.

You can use ::, the scope-resolution operator, to qualify a name with its namespace.

```cpp
Jack::pail = 12.34    // use a variable
Jill::Hill mole;      // create a type Hill structure
Jack::fetch();        // call a function
```

using Declarations and using Directives

A **using declaration** introduces only **one namespace member** at a time. Names introduced in a using declaration obey normal scope rules: They are visible from the point of the using declaration to the end of the scope in which the declaration appears. Entities with the same name defined in an outer scope are hidden.

variable declared in namespace Jill

```
namespace Jill{
    double bucket(double n){ ... }  // function definition
    double fetch;                   // variable declaration
    int pal;                        // variable declaration
    struct Hill{ ... };             // structure declaration
}
char fetch;
int main()
{
    using Jill::fetch;          // put fetch into local namespace
    double fetch;               // Error! Already have a local fetch
    cin >> fetch;               // read a value into Jill::fetch
    cin >> ::fetch;             // read a value into global fetch
    ...
}
```

global variable

Using declaration

Placing a using declaration at the external level adds the name to the global namespace:

```
void other();
namespace Jill{
    double bucket(double n){ ... }  // function definition
    double fetch;                   // variable declaration
    int pal;                        // variable declaration
    struct Hill{ ... };             // structure declaration
}
using Jill::fetch;          // put fetch into global namespace
int main()
{
    cin >> fetch;       // read a value into Jill::fetch
    other();
    ...
}

void other()
{
    cout << fetch;    // display Jill::fetch
    ...
}
```

A **using declaration**, makes a single name available. In contrast, the **using directive** makes all the names available.

```
using namespace Jack;    // make all the names in Jack available
```

```
#include <iostream>       // places names in namespace std
using namespace std;      // make names available globally
```

```
int main()
{
    using namespace Jack;  // make names available in main()
    ...
}
```

Generally speaking, the using declaration is safer to use than a using directive because it shows exactly what names you are making available.

```cpp
namespace sdm{
    const double BOOK_VERSION = 2.0;
    class Handle{...};
    Handle& getHandle();
}

void f1()
{
    using namespace sdm;

    cout << BOOK_VERSION;         // OK
    Handle h = getHandle();      // OK
}

void f2()
{
    using sdm::BOOK_VERSION;

    cout << BOOK_VERSION;       // OK
    Handle h = getHandle();   // Wrong
}

void f3()
{
    cout << sdm::BOOK_VERSION;    // OK

    double d = BOOK_VERSION;      // Wrong
    Handle h = getHandle();       // Wrong
}
```

2.3 Static library and dynamic library

When a C or C++ program is compiled, the compiler generates object code. After generating the object code, the compiler also invokes linker. One of the main tasks for linker is to make code of library functions available to your program.

Static Linking and Static Libraries (also known as an **archive**) is the result of the linker making copy of all used library functions to the executable file. Static Linking creates larger binary files, and need more space on disk and main memory. Examples of static libraries are, ***.a*** files in Linux and ***.lib*** files in Windows.

Dynamic linking and Dynamic Libraries Dynamic Linking doesn’t require the code to be copied, it is done by just placing name of the library in the binary file. The actual linking happens when the program is run, when both the binary file and the library are in memory. If multiple programs in the system link to the same dynamic link library, they all reference the library. Therefore, this library is shared by multiple programs and is called a "**shared library**" Examples of Dynamic libraries are, ***.so*** in Linux and ***.dll*** in Windows.

	advantages	disadvantages
Static Library	1. Make the executable has fewer dependencies, has been packaged into the executable file. 2. The link is completed in the compilation stage, and the code is loaded quickly during execution.	1. Make the executable file larger. 2. Being a library dependent on another library will result in redundant copies because it must be packaged with the target file. 3. Upgrade is not convenient and easy. The entire executable needs to be replaced and recompiled.
Dynamic Library	1.Dynamic library can achieve resource sharing between processes, there can be only one library file. 2. The upgrade procedure is simple, do not need to recompile.	1. Loading during runtime will slow down the execution speed of code. 2. Add program dependencies that must be accompanied by an executable file.

1.Building static libraries

Suppose we have three files as follows:

staticlib > hello.h > ...

```c
#ifndef HELLO_H_
#define HELLO_H_

void hello(const char *name);

#endif
```

staticlib > hello.c > hello(const char *)

```c
#include <stdio.h>

void hello(const char *name)
{
    printf("Hello %s!\n",name);
}
```

staticlib > main.c > main()

```c
#include "hello.h"

int main()
{
    hello("everyone");

    return 0;
}
```

Compile hello.c and main.c into main

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlib$ gcc -o main main.c hello.c
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlib$ ./main
Hello everyone!
```

the output

Run the program

Both static and dynamic libraries are created by **.o** files. Building a static library can following these steps:

Step1: Compile the library file into .o file. (We used gcc –o command and generate the executabl e file directly, that means we have no .o file.) If the .o file is existed, this step can be omitted.

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlib$ ls
hello.c  hello.h  main  main.c
```

No .o file is existed.

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlib$ gcc -c hello.c
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlib$ ls
hello.c  hello.h  hello.o  main  main.c
```

Build .o file

Step2: Build static libraries from .o files using **ar** command. Static libraries in linux are .a files whose name is like this **libxxx.a**, **lib** is a prefix, **.a** is the extension name.

cr flag tells the **ar** command to generate an archive file.

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlib$ ar -cr libmyhello.a hello.o
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlib$ ls
hello.c  hello.h  hello.o  libmyhello.a  main  main.c
```

Static library file

Step3: Use the static library in your program.

-L. tells compiler to find library in current directory

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlib$ gcc -o main main.c -L. -lmyhello
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlib$ ./main
Hello everyone!
```

-l. tells compiler to use library file “libmyhello.a”

- **-L**: indicates the directory of libraries
- **-l**: indicates the library name, the compiler can give the “**lib**” prefix to the library name and follows with **.a** as extension name.

You can specify the static library name when compiling.

Executable file

Specify the library name

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlib$ gcc -o hello main.c libmyhello.a
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlib$ ls
hello  hello.c  hello.h  hello.o  libmyhello.a  main  main.c
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlib$ ./hello
Hello everyone!
```

The public functions in the static library are already linked to the object file. If the static library is removed, the program can run normally.

Remove the library file

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlib$ rm libmyhello.a
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlib$ ls
hello  hello.c  hello.h  hello.o  main  main.c
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlib$ ./main
Hello everyone!
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlib$ ./hello
Hello everyone!
```

Run the program

You can bundle multiple object files in one static library by the following command.

ar –cr libtest.a test1.o test2.o

Create and use a static library in Visual Studio

https://docs.microsoft.com/en-us/cpp/build/walkthrough-creating-and-using-a-static-library-cpp?view=msvc-170

2.Building shared libraries

Suppose we have three files as follows:

function.h

sharedlib1 > function.h > ...

```cpp
#pragma once
void printHello();
```

function.cpp

sharedlib1 > function.cpp > ...

```cpp
#include <iostream>
#include "function.h"

void printHello()
{
    std::cout << "Hello World!" << std::endl;
}
```

main.cpp

sharedlib1 > main.cpp > ...

```cpp
#include "function.h"

int main()
{
    printHello();

    return 0;
}
```

Compile function.cpp and main.cpp into main

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/sharedlib1$ g++ -o main *.cpp
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/sharedlib1$ ./main
Hello World!
```

the output

Run the program

When building a shared library, remember to use the arguments "**-shared**" and "**-fPIC**". (pic, position independent code, means building a compiled program without relationship of address, its purpose for sharing among multi-programs).

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/sharedlib1$ g++ -shared -fPIC -o libfunction.so function.cpp
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/sharedlib1$ ls
function.cpp  function.h  libfunction.so  main.cpp
```

A shared library named libfunction.so is built

You can build a shared library in two steps:

Step 1: **g++ -fPIC -c function.cpp**

Step 2: **g++ -shared -o libfunction.so function.o**

Using a shared library

-L. tells compiler to find library in current directory

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/sharedlib1$ g++ -o main -L. main.cpp -lfunction
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/sharedlib1$ ls
function.cpp  function.h  libfunction.so  main  main.cpp
```

Executable file is built

-l. tells compiler to use library file "libfunction.so"

- **-L**: indicates the directory of libraries
- **-l**: indicates the library name, the compiler can give the "**lib**" prefix to the library name and follows with **.so** as extension name.

The running fails because now "main" relys on "libfunction.so". Most shared libraries in Linux are in **/lib** or **/usr/lib** folder, the operating system will search for that path by default. But our library is not in either of the folder, OS can not find the .so file for running main.

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/sharedlib1$ ./main
./main: error while loading shared libraries: libfunction.so: cannot open shared object file: No such file or directory
```

Using **export** command to set environment variable "**LD_LIBRARY_PATH**" and tell the OS where your library is.

Because the library is in the current path, set the variable to current path.

":" is the delimiter of paths

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/sharedlib1$ export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/sharedlib1$ echo $LD_LIBRARY_PATH
.:
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/sharedlib1$ ./main
Hello World!
```

Suppose we have five files as follows:

sharedlib2 > fun.h > ...

```cpp
#pragma once

void testa();
void testb();
void testc();
```

sharedlib2 > funa.cpp > ...

```cpp
#include <iostream>

void testa()
{
    std::cout << "This a testa." << std::endl;
}
```

sharedlib2 > funb.cpp > ...

```cpp
#include <iostream>

void testb()
{
    std::cout << "This is testb." << std::endl;
}
```

sharedlib2 > func.cpp > ...

```cpp
#include <iostream>

void testc()
{
    std::cout << "This is testc." << std::endl;
}
```

sharedlib2 > test.cpp > ...

```cpp

#include "fun.h"

int main()
{
    testa();
    testb();
    testc();

    return 0;
}
```

You can use "-fpic" before "-shared"

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/sharedlib2$ g++ funa.cpp funb.cpp func.cpp -fpic -shared -o libmyfun.so
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/sharedlib2$ ls
fun.h  funa.cpp  funb.cpp  func.cpp  libmyfun.so  test.cpp
```

Build one share library from three .cpp files

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/sharedlib2$ g++ test.cpp -L . -lmyfun -o mytest
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/sharedlib2$ ls
fun.h  funa.cpp  funb.cpp  func.cpp  libmyfun.so  mytest  test.cpp
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/sharedlib2$ ./mytest
./mytest: error while loading shared libraries: libmyfun.so: cannot open shared object file: No such file or directory
```

The program can not be run normally, because OS can not find the .so file while running the program.

```
sharedlib2
  mylib
    libmyfun.so
  fun.h
  funa.cpp
  funb.cpp
  func.cpp
  mytest
  test.cpp
```

This time we place the .so file into another directory.

Set **LD_LIBRARY_PATH** to that directory

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/sharedlib2$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.:./mylib
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/sharedlib2$ ./mytest
This a testa.
This is testb.
This is testc.
```

Run the program

Create and use a dynamic library in Visual Studio

https://docs.microsoft.com/en-us/cpp/build/walkthrough-creating-and-using-a-dynamic-link-library-cpp?view=msvc-170

3 Exercises

1. Define three functions that swap two values of integer, please use integer arguments, pointer arguments and reference arguments respectively. Write a test program to call these functions and display the result.

You are required to compile these functions into a shared library "libswap.so", and then compile and run your program with this shared library.

2. Write a three-file program based on the following namespace:

```
namespace SALES
{
    const int QUARTERS = 4;
    struct Sales
    {
        double sales[QUARTERS];
        double average;
        double max;
        double min;
    };

    // copies the lesser of 4 or n items from the array ar
    // to the sales member of s and computes and stores the
    // average, maximum, and minimum values of the entered items;
    // remaining elements of sales, if any, set to 0
    void setSales(Sales& s, const double ar[], int n);

    // gathers sales for 4 quarters interactively, stores them
    // in the sales member of s and computes and stores the
    // average, maximum, and minimum values
    void setSales(Sales& s);

    // display all information in structure s
    void showSales(const Sales& s);

}
```

The **first file** should be a header file that contains the namespace. The **second file** should be a source code file that extends the namespace to provide definitions for the three prototyped functions. The **third file** should declare two **Sales objects**. It should use the non-interactive version of setSales() to provide values for one structure and the interactive version of setSales() to provide values for the second structure. It should display the contents of both structures by using showSales().

A sample runs might look like this:

```
Non-interactive version of setSales() to provide values:
Sales:345.2   621.8   1073.5
Average:680.167
Max:1073.5
Min:345.2
Interactive version of setSales() to provide values:
Enter sales for 4 quarters: 459.3 902.1 1356.7 824.9
Sales:459.3   902.1   1356.7   824.9
Average:885.75
Max:1356.7
Min:459.3
```