Chapter 9 Memory Models and Namespaces

9.1 An overview

- The separate compilation of source files is useful when handling large programs, which typically consist of several source files.
- C++ offers many choices for storing data in memory. You have choices for
  - how long data remains in memory: storage duration;
  - which parts of a program have access to data: scope and linkage.
- You can allocate memory dynamically by using `new`, and placement `new` offers a variation on that technique.
- The namespace facility provides additional control over access.

9.2 Separate compilation

- You can compile source files separately into binary modules (object-code files) and then link them into the final executable program.
  - A C++ compiler compiles source files and typically, though not always, manages the linker.
  - If you modify just one file, you can recompile that file alone and then link it to the previously compiled versions of other files. This mechanism makes it convenient to manage large programs.
  - Furthermore, most C++ environments provide additional facilities to help with the management. Unix and Linux systems, for example, have `make` programs , which keep track of which files a program depends on and when they were last modified. If you run `make`, and it detects that you have changed one or more source files since the last compilation, `make` remembers the proper steps needed to reconstitute the program. IDEs also provide similar facilities.
- *Header* files: the files that contain common parts of the source code.
  - A header typically include: (1) function prototypes, (2) symbolic constants defined using `#define` or `const`, (3) structure/union/enumeration declarations, (4) class declarations, (5) template declarations, and (6) inline functions.
  - Using angle brackets `<>` to include a standard header file, and quotation marks `" "` to include your own header files. Upon seeing a bracket pair, the compiler searches the header in the standard header file directory; upon seeing a quotation mark pair, the compiler searches the header in the current working directory or the source code directory, and if it cannot find the header file, it then looks in the standard header file directory.

    > NOTE:
    >
    > a. you should not put a function definition in a header unless the it is an `inline` function, otherwise you may wind up encountering an error of function redefinition.
    >
    > b. you should not put a variable declaration in a header unless it is `const`.

  - You should include a header file only once. If you include one multiple times, the compiler will generate an error. But there is a syntax in the header to avoid error-generating, because it tells the compiler "if this file is already included, then do not include it again when seeing it again":

```
#ifndef HEADER_NAME_
#define HEADER_NAME_
... // header file's contents
#endif
```

- *Translation unit*s: the C++ standard uses the term *translation unit* instead of the term "file" in order to preserve greater generality, because the file metaphor is not the only possible way to organize information for a computer.
- Multiple Library Linking: the C++ standard allows each compiler designer the latitude to implement name decoration or mangling (see 8.4) as it sees fit, so you should be aware that binary modules (object-code files) created with different compilers will, most likely, not link properly. This name difference will prevent the linker from matching the function call generated by one compiler with the function definition generated by a second compiler. When attempting to link compiled modules, you should make sure that each object file or library was generated with the same compiler. If you are provided with the source code, you can usually resolve link errors by recompiling the source with your compiler.

9.3 Storage duration, scope, and linkage

- The various C++ storage choices are characterized by their (1) storage duration, (2) scope, and (3) linkage.
- As discussed in Chapter 4, there are three storage categories, or storage durations. Well, it is not complete. Actually, C++11 added the fourth. They are
  - **Automatic storage:** variables declared inside a function definition or a block, function arguments included, have automatic storage duration. They are created when the program execution enters the function or block in which they are defined, and the memory used for them is freed when the execution leaves the function or block. Automatic variables reside in a special block of memory called *stack*, where *First-in-last-out* (FILO) law rules. C++ has two kinds of automatic storage duration variables.

    > A function definition is itself a block, because it is enclosed by a `{}` pair.

  - **Static storage:** 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. C++ has three kinds of static storage duration variables.
  - **Dynamic storage:** memory acllocated by `new` operator persists until it is freed with the `delete` operator OR until the program ends, whichever comes first. These variables resides in a special block of memory called the *heap* or the *free store*.

    > Note that in some less robust operating systems, memory allocated bby `nre` operator persists even after the program is terminated, unless it is freed by `delete`.

  - **Thread storage (C++11):** this storage category is designed for multicore CPUs, which can handle several execution tasks simultaneously. Multicore processing allows a program to split computations into separate *thread*s that can be processed concurrently. Variables declared with keyword `thread_local` have storage that persists for as long as the containing thread lasts.
- Scope and linkage
  - *Scope* describes how widely visible a name is in a file (translation unit).

- *Linkage* describes how a name can be accessed in different units. A name with *external linkage* can be accessed across files, and a name with *internal linkage* can only be shared by functions within a single file.

  > Names of automatic variables have no linkage because they are not shared.

- A C++ variable can have one of several scopes.
  - A variable that has *local scope*, also termed *block scope*, is known ONLY within the block in which it is declared.
  - A variable that has *global scope*, also termed *file scope*, is known throughout the file after the point where it is declared.
  - Names used in a *function prototype scope* are known just within the parentheses enclosing the argument list - that is why it does not really matter what they are or if they are present.
  - Members declared in a class have *class scope*.
  - Variables declared in a namespace have *namespace scope*. The global scope is a special case of namespace scope.

  > Note:
  >
  > a. automatic variables have local scope, and a static variable can have either scope, depending on how it is defined.
  >
  > b. functions can have class scope or namespace scope, including scope. But a function cannot have local scope - is a function were to have local scope, it could only be known to itself and hence could not be called.

- Automatic storage duration variables: largely omitted
  - *Register variable*s: C originally introduced the `register` keyword to suggest rhat the compiler use a CPU register, rather than a memory block, to store an automatic varibale: `register int a;`. The idea was that this would allow faster access to the variable. In C++, the hint was generalized to mean that the variable was heavily used and perhaps the compiler could provide some sort of special treatment. Since C++, however,even that hint is deprecated, leaving `register` as just a way to explicitly identify a variable as being automatic, assuming the former role of `auto`.
- Static duration variables
  - C++, like C, provides static storage duration variables with three kinds of linkage:

linkage	**accessibility**	**scope**	**how to declare**
external linkage	across files	file	declare it outside functions (in other files, prefix it with `extern`)
internal linkage	within a single file	file	declare it outside functions, and prefix it with `static`
no linkage	to one function or one block	block	declare it inside a function or block, and prefix it with `static`

  > Note that the keyword `static` has somewhat different meanings in the two uses shown above. When used within a block to indicate a static variable with no linkage, `static` indicates the kind of storage duration. When used with a declaration outside of any block, `static` indicates internal linkage; the variable already has static duration. One may term

> this *keyword overloading*, with the precise meaning determined by context.

- Static duration variables last for the duration of the program, so they are less ephemeral than automatic variables.
- The program allocates a fixed block of memory to hold all the static variables.
- If you do not explicitly initialize a static variable, the compiler implicitly sets it to 0 - a privilege an automatic variable do not enjoy. This initialization is called *zero-initialization*. Otherwise, if a static variable is explicitly initialized by a constant expression, it is *constant-expression initialization*; if a static variable is explicitly initialized by a non-constant expression, it is *dynamic initialization*.

  > Side notes:
  >
  > A *constant expression* is an expression whose value cannot change and that can be evaluated at compile time. For example, a literal such as `5L` or `'a'` is a constant expression; `x` declared in `const int x = 0;` is also a constant expression. But `y` declared in `int y = 0;` is NOT, for `y`'s value might change. `z` declared in `const z = get();` is NOT, for `z`'s value is unknown until runtime. If an expression is not a constant expression, it cannot be applied in some situations, say, specifying the size of an array.
  >
  > How do we ensure a variable is a constant expression? C++11 introduced a now keyword, `constexpr`, to let us ask the compiler to verify that a variable is a constant expression by declaring the variable in a `constexpr` declaration. Variables declared this way are implicitly `const` and must be initialized by constant expressions:
  >
  > ```
  > 1  constexpr int m = 20;       // ok. 'm' is a constant expression
  > 2  constexpr int l = m + 1;    // ok. 'l' is a constant expression
  > 3  constexpr int *q = nullptr; // ok. 'q' is a constant expression
  > 4  constexpr int k = size();   // ok ONLY IF "size" is a
  > 5                              // constexpr function
  > ```
  >
  > For more detailed discussion on `constexpr` such as `constexpr` functions, see *C++ Primer*.

- Static duration, external linkage: accessible across files, declared outside functions.
  - Variables with external linkage are often simply referred to as *external variable*s.
  - They necessarily have static storage duration and file scope.
  - External variables must be declared in each file that uses the variable. To allocate a block of memory to store a variable, you use a *defining declaration* (the normal way of declaring a variable); to tell the compiler a variable in a file is essentially referred to an existing variable in another file, you use a *referencing declaration* (has keyword `extern` at the front). Note: Any variable's defining declaration can only appear ONCE in all files (*one definition rule*). Example:

```
// File #1, outside any blocks
double a;           // defining declaration
extern double b;    // referencing decalration
double c;
double d;
// File #2, outside any blocks
extern double a;    // referencing declaration
double b;           // defining declaration
double c;           // error. the linker is confused
int d;              // ok.
```

In the example above, the linker (not the compiler) will generate a linker error (not a compile error) massage because you do not use `extern` for `c` in file #2 (or file #1) - the linker gets confused when you attempt to allocate two blocks of memory that are denoted by the same name `c` (or more precisely, the same name of the same type), without telling the linker whether or not they are essentially different variables. As for `d`, there is no error because the linker knows from their types that they are two different variables.

> If two cognominal variables in two files are essentially the same variable, then this variable is an external variable, as discussed above; if they are two unrelated variable, it is a different story, as discussed later.
>
> Plus, in my IDE (VS 2015), the decorated names for `double c`, `double d` and `int d` are `?c@@3HA`, `?d@@3HA` and `?d@@3NA`, respectively.

- In C++ (not C), the `const` modifier alters the default storage classes slightly: whereas a regular external variable has external linkage, a `const` external variable has internal linkage by default unless the defining declaration is prefixed by `extern`, like referencing declarations.

  > This alteration makes `const` variable able to be declared in header files. Without the alteration, there would be a linker error because after preprocessing, the defining declaration of the variable which is declared in a header appears in every files that includes the header - a violation of the one definition rule.
  >
  > By the way, You should NOT declare a non-`const` variable in a header, otherwise there would be a linker error caused by violating the one definition rule.

- Static duration, internal linkage: accessible to one file, declared outside functions with `static`.
  - Applying the `static` modifier to a file-scope variable gives it internal linkage. A variable with internal linkage is local to the file that contains it, yet a regular external variable has external linkage, meaning that it can be used in different files with keyword `extern`, as discussed above.

```
// File #1, outside any blocks
static double a;  // known only within file #1
double b;         // external variable
double c;         // external variable
// File #2, outside any blocks
static double a;  // known only within file #2
static double b;  // known only within file #2
extern double c;  // refer to 'c' in file #1
// File #3, outside any blocks
static double a;  // known only within file #3
extern double b;  // refer to 'b' in file #1
```

    - A internal-linkage variable overrides the cognominal external-linkage variable.
- Static duration, no linkage: accessible to one block, declared inside a block with `static`.
    - A variable declared within a block (a function is also a block) is *local variable*. Applying `static` to a local variable's declaration gives it static storage (automatic storage, otherwise).
    - A local variable with static storage is known only to the block in which it is declared, but persists in memory until the whole program terminates - which means it exists even while the block is inactive.
    - Local variables with static storage can preserve their values between function/block calls, so they are useful for reincarnation. Also, they are only initialized once; subsequent calls to that function/block do not reinitialize it - the program just skips the initialization statement.

```
// inside a function/block
static a = 100; // an initialization. Executed only once.
a = 101;        // an ordinary assignment. Executed every time.
```

> You should distinguish "initialization" and "assignment". An "initialization" is assignment that accompanies a declaration, such as `int k = 1;`.
>
> ```
> int k = 1;      // an initialization.
> k = 1;          // an ordinary assignment, NOT an initialization
> ```

- Storage-related keywords - storage class specifiers and cv-qualifiers
    - Certain C++ keywords, called *storage class specifiers* and *cv-qualifiers*, provide additional information about storage.
        - storage class specifiers: `auto` (eliminated as a specifier in C++11), `register`, `static`, `extern`, `thread_local` (added by C++11), `mutable`.
        - cv-qualifiers: `const`, `volatile`.
    - Storage class specifiers:
        - `auto` - once used to indicate automatic storage but is now repurposed. Since C++11, it is used for automatic type deduction and trailing return type syntax.
        - `register` - once used to indicate register variables but is now generalized. Since C++11, it is used for explicitly indicating automatic storage.
        - `static` - when used with a file-scope declaration (static storage already), it indicates internal linkage; when used with a block-scope declaration, it indicates static storage.

        - `extern` - used to indicate a reference declaration of an external variable.
        - `thread_local` - used to indicate a `thread_local` variable, which is to a thread much as a regular static variable is to the whole program.
        - `mutable` - used to indicate that a particular member of a structure or class can be altered even if a particular structure or class variable is a `const`.

```
struct data {char name[30]; mutable int index;};
const data s = {"Lawrence Berkeley Lab", 1};
strcpy(s.name, "LBL");     // not allowed
s.index++;                 // allowed
```

    - cv-specifiers:
        - `const` - used to indicate that memory, after initialized, should not be altered by a program.
        - `volatile` - used to indicate that the value of a variable can be altered even though nothing in the program code modifies the contents - the source of modification might be the hardware, another program, or something else other than the program itself. If the compiler notices that a particular variable is used by the source code twice within a few statements. Rather than have the program look up the value twice, the compiler might cache the value in a register. This optimization assumes that the value of the variable does not change between the two uses. Unless you declare a variable as volatile, the compiler can feel free to make such optimization.

9.4 Functions and linkage

- Like variables, functions also have linkage properties, although the selection is more limited than for variables.
- C++, like C, does not allow you to define a function inside another. So there is no automatic storage for functions.

    > In short, no nested-functions. But some programming languages allows nested-functions, like MATLAB and Python.

- Storage: all functions have static storage duration, meaning they are all present as long as the program is running.
- Linkage: all functions have external linkage by default, meaning they can be shared across files.
    - Like external variables, a function prototype, a.k.a. function declaration, should appear before the function is called, in any file (translation unit).
    - If you want to call a function in file A but that function is defined in file B, you can use `extern` to the function prototype in file A. But it is optional.
    - You can use `static` to give a function internal linkage. You should prefix `static` in the function's both prototype and definition. That means the function is only to one file. Like with variables, a internal-linkage function overrides the cognominal external-linkage function.
- Inline functions: inline functions are excepted from the one definition rule so it is allowed to place inline function definitions in a header file. Thus, each file that includes the header file ends up having the inline function definition. However, C++ does requires that all the inline functions for a particular function be identical.
- Where does the compiler and linker find functions?

- Suppose you call a function in a file in a program. If the function prototype in that file indicates the function is `static`, the compiler looks only in that file for the function's definition. Otherwise, the compiler, together with the linker, looks in all files within the program.
- If two definitions are found, the compiler sends you an error because you can have only one definition in these files for an external function. If no definition are found, the compiler searches the libraries. This mechanism implies that if you have a function that has the same name as a library function, the compiler picks your version rather than the library version.
- Some compiler-linkers need explicit instructions to identify which libraries to search. This is especially the case when your compiler and linker are not integrated into an IDE.

9.5 Language linking

- *Language linking* is another form of linking that affects functions.
- Two linking approaches
  - C does not allow overloading, so for internal purposes a C compiler might translate a C function such as `func` to `_func` (name decoration). This is called *C language linkage*.
  - However, C++ allows overloading so the same C++ function name has to be translated into separate symbolic names. For instance, a C++ compiler might translate `func(int)` and `func(int, double)` to `_func_i` and `func_i_d`. This approach is termed *C++ language linkage*.
- From the discussion above, C and C++ compilers use different conventions to decorate function names. So when the linker looks for a function to match a C++ function call, it has to use a different search method than it does to match a C function call. You can use function prototypes to tell the linker the called function is a C function or a C++ function:

```
1 extern "C" void func1(int);   // use C convention for name search
2 extern void func2(int);       // use C++ convention for name search
3 extern "C++" void func3(int); // use C++ convention for name search, too
```

  These *C and C++ language linkage specifier*s are required by the C++ standard. But implementation may provide additional language linkage specifiers.

9.6 More about dynamic storage

- Basic usage of `new` and `delete`: see Chapter 4.
- When `new` fails: it may be that `new` cannot find the requested amount of memory. For its first decade, C++ handles that eventuality by returning a null pointer. Currently, however, it throws a `std::bad_alloc` exception. See Chapter 15.
- Allocation functions and deallocation functions
  - The `new` and `new[]` operators call upon two functions:

    `void *operator new(std::size_t);` and `void *operator new[](std::size_t);`

    These are termed *allocation function*s, and they are part of the global namespace. Similarly, the `delete` and `delete[]` call upon *deallocation function*s:

    `void operator delete(void *);` and `void operator delete[](void *);`
  - So using `new`, `new[]` and `delete`, `delete[]` like functions are permitted. Using these operators is calling their corresponding functions, in fact.

```
int *pi1 = new int;              int *pi2 = new(sizeof(int));
int *pa1 = new int[10];          int *pa2 = new(10 * sizeof(int));
delete pi;                       delete(pi);
delete[]pa1;                     delete[](pa2);
```

  - These functions are *replaceable*. That means if can supply replacement functions for these four operators and tailor the functions to meet you specific demands.
    - One option, for instance, is to define replacement functions with class scope so that they can be tailored to fit the allocation needs of a particular class. Your would use the `new` operator as usual, but the new operator would call upon the replacement `new()` function.

- The placement `new` operator
  - Normally, the `new` operator has the responsibility of finding in the heap a block of memory that is large enough to handle the amount of memory you request.
  - A variation, called *placement* `new`, allowed you to specify the location to be used. To use this feature, the header `<new>` should be included, which provides a function prototype for this version of `new`.
  - Placement `new` can be used
    - to set up user-designed memory-management procedures,
    - to deal with hardware that is accessed via a particular address,
    - to construct objects in a particular memory location.
  - To use placement `new` / `new[]`, you should provide an argument that indicates the intended starting address: `new (startAddr) type` and `new (startAddr) type[size]`. Note that it is your responsibility, no longer the program's, to keep track of whether the location has already been used. For example

```
// regular forms of 'new': on heap
int *p1 = new int; int *p2 = new int[10]; int *p3 = new structType;
// placement 'new' (the header <new> should be included)
char buffer1[50];  char buffer2[200]; const int N = 10;
int *p4 = new (buffer1) int;                          // use buffer1
int *p5 = new (buffer2) int[N];                       // use buffer2
int *p6 = new (buffer2 + N * sizeof(int)) structType; // use buffer2
```

```
char buffer3[200] = "abcdefg"; const int N = 10;
int *q1 = new (buffer3) int[N]; // overwrite old data "abcdefg"
int *q2 = new (buffer3) int[N]; // overwrite q1[0] ~ q1[9]
double *q3 = new (buffer3 + 9*sizeof(int)) double; // overwrite q2[9]
```

  - To provide more flexibility, C++ allows programmers to overload placement `new`.
  - `delete` and `delete[]` can ONLY be used to free memory on the heap. If the memory allocated by placement `new` / `new[]` is somewhere else, say, in the static memory or on the stack, these operators cannot be applied; otherwise there will be a runtime error.

    > In short, the jurisdiction of `delete` / `delete[]` is confined to the heap.

```
char buffer1[100];
char *buffer2 = new char[100]; // buffer2 is on the heap
p1 = new (buffer1) double[9];       // not on the heap
p2 = new (buffer2) double[9];       // on the heap (placement 'new')
p3 = new double[9];                 // on the heap (regular 'new')
delete[]p1;               // runtime error
delete[]p2;  delete[]p3;  // ok
```

- Like the regular `new`, placement `new` has its corresponding functions so you can use it as an operator or as a function.

```
int *pi1 = new (ptrBox1) int;
int *pi2 = new(sizeof(int), ptrBox2);
int *pa1 = new (ptrJar1) int[10];
int *pa2 = new(10 * sizeof(int), ptrJar2);
```

9.7 Namespaces

- *Namespace problem*s: problems that caused by incompatible naming. For instance, two third-party libraries might both define a `List` class in different ways.
- Some terms:
  - A *declarative region* for a variable is a region in which the declaration can be made. For example, the declarative region for a automatic variable is the block in which it is declared.
  - The *potential scope* for a variable begins at its declaration statement and extends to the end of its declarative region. Therefore, a variable's potential scope is more limited than its declarative region since you cannot use a variable before it is declared.
  - However, a variable might NOT be visible everywhere in its potential scope. For instance, it might be hidden by another variable of the same name declared in a nested declarative region. The portion of the program that can actually see the variable is termed the *scope* of that variable.

    > For a given variable, scope ⊂ potential scope ⊂ declarative region

- Traditional namespaces
  - C++'s (and C's) rules about variables' declarative regions constitute a kind of namespace hierarchy: Names of variables used in one declarative region do not conflict the same names in another declarative region.
  - Declarative regions can be nested. For example

```
int main()                // ----------------------+
{                               //                |
  int i = 100;                  //                |
  for (int i = 0; i < 10; i++){ // --+     declarative region
     //...        nested declarative region       |
  }                             // --+             |
}                         // ----------------------+
```

The `i` inside the loop (a block) does not conflict with the `i` declared outside. A local variables override the cognominal global variable, as is shown in the code above.

- New namespace features: *named namespace*s
  - C++ adds the ability to create *named namespace*s, sometimes shortly referred to as *namespace*s, by defining a new kind of declarative region, one whose main purpose is to provide an area in which to declare names. Names used in this namespace do not conflict with the same names declared in other namespaces.
  - Namespaces are created with keyword `namespace`. When declaring a name in a namespace, its identity (variable or function) and types should be specified. For example,

```
namespace TheReach{
  int greatLord;                // declare a variable
  double liegeLords;            // declare a variable
  int tax();                    // declare a function
  char theOldTown[10];          // declare an array
} // no semicolon (unlike structures, classes, etc.)
namespace TheNorth{
  char greatLord;               // declare a variable
  struct liegeLords {...};      // declare a structure type
  int tax;                      // declare a variable
  void theWall(double);         // declare a function
} // no semicolon (unlike structures, classes, etc.)
```

  - Namespaces can be located at the global level or inside other namespaces, but they cannot be placed in a block.
    - Thus, A name declared in a namespace has external linkage by default, unless it refers to a constant (in that case, the name has internal linkage, per the discussion above).
  - In addition to user-defined namespaces, there is a namespace called the *global namespace*, or *gloabal declative region*. This corresponds to the file-level declarative region, where *global variable*s are declared.
  - Namespaces are *open*, meaning that you can add names to existing namespaces. For example, the code below adds the name `theBearIsland` to the aforementioned user-defined namespace `TheNorth`:

```
namespace TheNorth{
  char *theBearIsland(const int *);
}
```

  - You can provide the function definition for a function declared in a namespace by using keyword `namespace`, in the same file OR in another file.

```
namespace TheNorth{
  void theWall(double n) // function body capsulated in a namespace
  {
    //...
  }
}
```

    You can also provide the function definition in another way.

```cpp
void TheNorth::theWall(double n) // less capsulated
{
  //...
}
```

- The *scope-resolution operator* `::` can be used to access a name in a namespace, qualifying a name with its namespace. Names adorned with its namespace is termed the *qualified name*, otherwise the *unqualified name*.

```cpp
TheReach::greatLord = 1; TheNorth::greatLord = 'a';
int t = TheReach::tax();
```

- `using` declarations and `using` directives
  - Having to qualify names every time they are used is sometimes unpleasant. So C++ provides two alternative approaches: the `using` declaration and the `using` directives.
  - The `using` declaration: `using Name::name;`
    - This declaration adds a particular name to your intended declarative region. After doing so, you cannot declare another variable that has the same name.

```cpp
namespace Alice{double a; bool b;}
char a;
using Alice::b;    // add 'b' to the global declarative region
bool b;            // error. 'b' already exists
int main()
{
  using Alice::a;  // add 'a' to this local declarative region
  double a;        // error. 'a' already exists
  bool b;          // ok. Override the global 'b'.
  std::cin >> a;   // read a value into Alice::a
  std::cin >> ::a; // read a value into the gloabal a
}
```

    - In the code above, like any other local variable, `a` override the global variable by the same name. If you want to overturn the overriding, you should use `::a` to access the global one.
    - The compiler will generates an error if you have code like this because of possible name conflicts:

```cpp
using TheReach::greatLord; using TheNorth::greatLord;
```

      For instance, if you have a statement `greatLord++;`, the compiler does not know which `greatLoard` you mean.
  - The `using` directive: `using namespace Name;`
    - A `using` declaration makes a single name available in a declarative region. In contrast, a `using` directive makes all the names available.
    - Placing a `using` directive at the global declarative region makes the namespace's names available globally. For example,

```
#include ...
using namespace std; using namespace Alice;
...
```

> This is an alternative of using external variables.

- Placing a `using` directive inside a function makes the names avaible just in that function. For example,

```
int main()
{
  using namespace std; using namespace Alice;
  ...
}
```

- Using the `using` directive to import all the names from a namespace wholesale is NOT the same as using multiple `using` declarations.
  - Suppose a namespace and a declarative region both define the same name.
  - If you attempt to using a `using` declaration to bring the namespace name into the declarative region, you will get an error because of names conflict.
  - If you use a `using` directive to bring the namespace name into the declarative region, the local version of the name override the namespace version. You can overturn the overriding by explicitly using the namespace-resolution operator `::`.
  - It is better to prefer the `using` declaration over the `using` directive, because the former is more specific about which name you want to import.

```
namespace Alice {double a; bool b;}
namespace Bob {int k; struct Stock{...};}
char k;         // global variables
using Alice::b; // global variables
void do_sth(); // function prototypes

int main(){
  using Alice::a;
  int a[10];             // error. Names conflict
  using namespace Bob;
  Stock aweDress;        // a type Bob::Stock structure variable
  double k;              // NOT an error. It just hides Bob::k
  std::cin >> k;         // read a value into the local 'k'
  std::cin >> ::k;       // read a value into the global 'k'
  std::cin >> Bob::k;    // read a value into Bob::k
  do_sth();
}

void do_sth(){
  Stock vigorMusic;      // error. 'Stock' undefined
  Bob::Stock vigorMusic; // ok. a type Bob::Stock structure variable
  b = true;              // ok. Alice::b is declared globally
}
```

- More namespace features
  - Namespaces can be nested, like this

```
namespace world{
  namespace france{int paris; int nice; double marseille;}
  namespace us{
    namespace louisiana{float paris; long batonRouge;}
    namespace idaho{short paris;}
  }
  char peace;
}
```

    And then you can import them like this

```
using namespace world::france;     // import a namespace wholesale
using namespace world::us::idaho;  // import a namespace wholesale
using world::us::louisiana::paris; // import a single name
using world::peace;                // import a namespace wholesale
```

  - You can use `using` directives/declarations inside namespaces, like this

```
namespace myth{
  using world::peace;
  using namespace world::france;
  using std::cin; using std::cout;
  char c;
}
```

    You use names like this

```
std::cin >> myth::c;
std::cin >> myth::peace;
std::cin >> myth::marseille;
std::cin >> world::france::paris; // same as myth::paris
```

    or like this

```
using namespace myth;
cin >> peace;
cin >> paris;
```

  - You can create an alias for a namespace, like this

```
namespace legend = myth;
namespace world_france = world::france;
```

  - You can create an *unnamed namespace*, like this

```
namespace
{
  int ice;
  double cream;
}
```

This code behaves as if it were immediately followed by a `using` directive, so an unnamed namespace can only be applied to the file in which it is defined.

- So this technique provides an alternative to using static variables with internal linkage:

```
namespace{int cnt;}
int main(){
  //...
}
```

has the same effect as

```
static int cnt;
int main(){
  //...
}
```

- Multi-file program.
  - The definition of user-defined namespaces can be stored in a user-defined header, say, `namesp.h`.
  - Functions declared in these namespaces can be defined in a source file, say, `namesp_fun.cpp`.
  - Therefore, in a source file, say, `driver.cpp`, which uses the namespace, you just have to include the header.
  - Doing so makes it possible to compile `driver.cpp` and `namesp_fun.cpp` separately. Changing `namesp_fun.cpp` does not need to recompile `driver.cpp`.

```
// file: namesp.h
#ifndef NAMESP_H_
#define NAMESP_H_
#include <string>
namespace pers {
  int index;
  struct Person{std::string fname; std::string lname;};
  void getPerson(Person &);
  void showPerson(const Person &);
}
namespace debts {
  using namespace pers;
  struct Debt{Person name; double amount;};
  void getDebt(Debt &);
  void showDebt(const Debt &);
}
#endif
```

```
// file: namesp_fun.cpp (Version #1) // more capsulated
#include <iostream>
#include "namesp.h"
namespace pers{
  void getPerson(Person &rp) {...; std::cin >> ...; ...}
  void showPerson(const Person &rp) {...; std::cout <<...; ...}
}
namespace debts{
  void getDebt(Debt &rd) { getPerson(rd.name); std::cin >> ...; ...}
  void showDebt(const Debt &rd) {...; std::cout << ...; ...}
}
```

```
// file: namesp_fun.cpp (Version #2) // less capsulated
#include <iostream>
#include "namesp.h"
void pers::getPerson(Person &rp) {...; std::cin >> ...; ...}
void pers::showPerson(const Person &rp) {...; std::cout <<...; ...}
void debs::getDebt(Debt &rd)
{ getPerson(rd.name); std::cin >> ...; ...}
void debts::showDebt(const Debt &rd)
{...; std::cout << ...; ...}
```

```
// file: driver.cpp
#include "namesp.h"
void another(void);

int main(void){
  using debts::Debt; using debts::showDebt;
  Debt golf = {{"Emma", "Roberts"}, 100.0;};
  showDebt(golf);
  another();
}
void another(void){
  using pers::Person;
  Person collector = {"John", "Kaine"};
  pers::showPerson(collector);
}
```

9.8 Namespaces and common programming idioms

- As programmers become more familiar with namespaces, common programming idioms for large projects emerge. Here are some current guidelines:
  - Use variables in a named namespace instead of using external global variables.
  - Use variables in an unnamed namespace instead of using static global variables.
  - If you develop a third-party library of functions or classes, place them in a namespace.

> C++'s standard library functions are placed in the standard namespace `std`. This extends to functions brought in from C. For example, the `math.h` header, which is C-compatible, does not use namespaces, but the C++ `<cmath>` header places math library functions in the `std` namespace.

- Use the `using` directive ONLY as a temporary means of converting old code to namespace usage. You should prefer the `using` declaration because it is more specific.
- Do NOT use `using` directives in header files. Because doing so conceals which names are being made available, and the ordering of header files may affect behavior.
- Place a `using` directive after all the preprocessor `#include` directives.
- Preferentially import names by using the scope-resolution operator or a `using` declaration, instead of a `using` directive.

  > Recommended: `std::cout` or `using std::cout;`
  >
  > Not recommended: `using namespace std;` (a `using` directive)

- Preferentially use local scope, instead of global scope, for `using` declarations.
- Place namespace definitions in a header. Place the function definitions for functions declared in a namespace in other source files.

□