Thinking in C++, 2nd ed. Volume 1

Ch. 10: Name Control

Static elements from C

The static keyword was overloaded in C before people knew what the term “overload” meant, and C++ has added yet another meaning.
The underlying concept with all uses of static seems to be “something that holds its position” (like static electricity), whether that means a physical location in memory or visibility within a file.

Static elements from C

In both C and C++ the keyword static has two basic meanings:

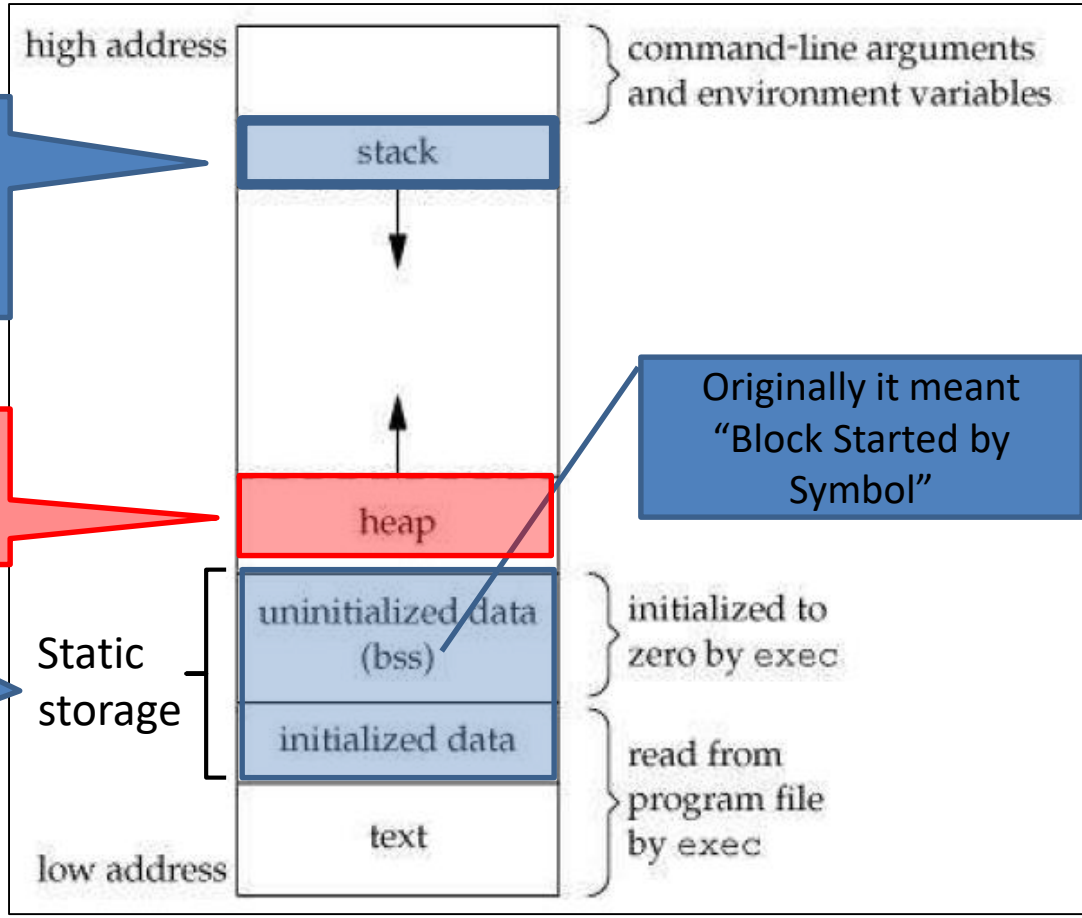
- Allocated once at a fixed address; that is, the object is created in a special static data area (a.k.a. **static storage**) rather than on the stack each time a function is called.
  - See next slide
- Local to a particular translation unit (and local to a class scope in C++, as you will see later). Here, static controls the visibility (a.k.a. **scope**) of a name, so that name cannot be seen outside the translation unit or class.
  - This also describes the concept of **linkage**, which determines what names the linker will see.

Static storage

Contains:
- Local **auto** objects

Contains:
- Objects allocated dynamically

Contains:
- Local, static and constant objects
- Unitialized global objects



Remember: In C (and C++) uninitialized **static** variables are initialized by default with 0.

Source: Leo Ferres, Memory management in C: The heap and the stack

Static elements from C

Review the next part of this section until

Other storage class specifiers

Namespaces

In a large project, lack of control over the global name space can cause problems. To solve these problems for classes, vendors often create long complicated names that are unlikely to clash, but then you're stuck typing those names.

It's not an elegant, language-supported solution.

Namespaces

You can subdivide the global name space into more manageable pieces using the namespace feature of C++.

The **namespace** keyword, similar to class, struct, enum, and union, puts the names of its members in a distinct space.

Namespaces

A namespace definition can appear only at global scope, or nested within another namespace.

No terminating semicolon is necessary after the closing brace of a namespace definition.

A namespace definition can be “continued” over multiple header files using a syntax that, for a class, would appear to be a redefinition:

```
//: C10:Header1.h
#ifndef HEADER1_H
#define HEADER1_H
namespace MyLib {
  extern int x;
  void f();
  // ...
}

#endif // HEADER1_H ///:~
//: C10:Header2.h
#ifndef HEADER2_H
#define HEADER2_H
#include "Header1.h"
// Add more names to MyLib
namespace MyLib { // NOT a redefinition!
  extern int y;
  void g();
  // ...
}

#endif // HEADER2_H ///:~
//: C10:Continuation.cpp
#include "Header2.h"
int main() {} ///:~
```

A namespace name can be *aliased* to another name, so you don't have to type an unwieldy name created by a library vendor:

```
//: C10:BobsSuperDuperLibrary.cpp
namespace BobsSuperDuperLibrary {
  class Widget { /* ... */ };
  class Poppit { /* ... */ };
  // ...
}
// Too much to type! I'll alias it:
namespace Bob = BobsSuperDuperLibrary;
int main() {} ///:~
```

Static members in C++

Sometimes we need a single storage space to be used by all objects of a class. We could use a ***global variable***, but this is not very safe:

- Global data can be modified by anyone
- Its name can clash with other identical names in a large project.

It would be ideal if the data could be stored **as if** it were global, but be **hidden** inside a class, and clearly associated with that class.

example on next slide

```
class A {
  static int i;
public:
  //...
};


int A::i = 1;
```

The initialization of a static member must be done outside of the class declaration, in the definitions section (file)!

Important: Do not confuse with the static **constants** from Ch.8!

```
class StringStack {
  static const int size = 100;
  const string* stack[size];
  int index;
public:
  StringStack();
```

Initialized at the point of definition!

Actually, even the static integer **constants** from Ch.8 can be defined outside the class (It is consistent with ODR!):

```
class StringStack {
  static const int size = 100;
  const string* stack[size];
  int index;
public:
  StringStack();
```

Initialized at the point of definition!

```
class A{
  static const int size;
};

const int A::size = 42;
```

QUIZ

Define a class **Turtle** with a static data member named **pop** (population), initialized with zero. The constructor increments **pop**.

```
class Turtle{
    static unsigned pop;
public:
    Turtle(){pop++;}
    unsigned getPop(){return pop;}
};

unsigned Turtle::pop = 0;

int main(){
    Turtle t1, t2, t3;
    cout << t1.getPop() << endl;
    cout << t2.getPop() << endl;
    cout << t3.getPop() << endl;
}
```

```
3
3
3
```

Static arrays (const or not)

With static consts of integral types you can provide the definitions inside the class, but **for everything else** (including **arrays** of integral types, even if they are static const) **you must provide a single external definition**.

example on next slide

```
//: C10:StaticArray.cpp
// Initializing static arrays in classes
class Values {
  // static consts are initialized in-place:
  static const int scSize = 100;
  static const long scLong = 100;
  // Automatic counting works with static arrays.
  // Arrays, Non-integral and non-const statics
  // must be initialized externally:
  static const int scInts[];
  static const long scLongs[];
  static const float scTable[];
  static const char scLetters[];
  static int size;
  static const float scFloat;
  static float table[];
  static char letters[];
};
```

```
int Values::size = 100;
const float Values::scFloat = 1.1;

const int Values::scInts[] = {
  99, 47, 33, 11, 7
};

const long Values::scLongs[] = {
  99, 47, 33, 11, 7
};

const float Values::scTable[] = {
  1.1, 2.2, 3.3, 4.4
};

const char Values::scLetters[] = {
  'a', 'b', 'c', 'd', 'e',
  'f', 'g', 'h', 'i', 'j'
};
```

Note that the **static** keyword is not repeated at initialization!

```
float Values::table[4] = {
  1.1, 2.2, 3.3, 4.4
};

char Values::letters[10] = {
  'a', 'b', 'c', 'd', 'e',
  'f', 'g', 'h', 'i', 'j'
};
```

Note: C++11 has introduced the related qualifier **constexpr**, which allows more definitions inside the class:

```
//: C10:StaticArray.cpp
// Initializing static arrays in classes
class Values {
  // static consts are initialized in-place:
  static const int scSize = 100;
  static const long scLong = 100;
  // Automatic counting works with static arrays.
  // Arrays, Non-integral and non-const statics
  // must be initialized externally:
  static const int scInts[];
  static const long scLongs[];
  static const float scTable[];
  static const char scLetters[];
  static int size;
  static const float scFloat;
  static float table[];
  static char letters[];
};
```

```
class A{
  static constexpr int scInts[2] = {1, 2};
  static constexpr float scFloat = 1.5;
};
```

SKIP

Nested and local classes

Static member functions

static member functions, like **static** data members, work for the class as a whole rather than for a particular object of a class. Instead of making a global function that lives in and "pollutes" the global or local namespace, you bring the function inside the class.

```
//: C10:SimpleStaticMemberFunction.cpp
class X {
public:
  static void f(){};
};
```

Static member functions

You can call a static member function in the ordinary way, with the dot or the arrow, in association with an object. However, it's more typical to call it by itself, without any specific object, using the scope-resolution operator:

```
//: C10:SimpleStaticMemberFunction.cpp
class X {
public:
  static void f(){};
};

int main() {
  X::f();
} ///:~
```

Static member functions

- Cannot access ordinary data members, only static data members
- Can call only other static member functions

Normally, the address of the current object (**this**) is quietly passed in when any member function is called

- but a static member has no **this** (which is the reason it cannot access ordinary members)
- Thus, we get the tiny increase in speed afforded by a global function because a static member function doesn't have the extra overhead of passing **this**. At the same time you get the benefits of having the function inside the class.

Static member function example

```
class Turtle{
    public:
static int getPop(){cout <<this;};
};

int main() {
    Turtle t;
    t.getPop();
}
```

```
prog.cpp:5:28: error: ‘this’ is unavailable for static member functions
 static int getPop(){cout <<this;};
                           ^~~~
```

QUIZ: Is this code correct?

```
#include <iostream>

using namespace std;

class Foo{
  static int i;
public:
  static void printi(){cout <<i <<endl;}
};

int Foo::i = 42;

int main(){
   Foo::printi();
}
```

Solution: Yes. Although the function call in the main program looks weird, it is legal for a static member function!

```
#include <iostream>

using namespace std;

class Foo{
  static int i;
public:
  static void printi(){cout <<i <<endl;}
};

int Foo::i = 42;

int main(){
   Foo::printi();
}
```

SKIP

Static initialization dependency
Alternate linkage specifications

No homework assigned for this chapter!

Individual work (preparation for midterm):

- **End-of-chapter 2, 3, 4, 16**