Topic 1: Static members of classes
Topic 2: Namespaces

Comp Sci 1570 Introduction to C++

Outline

1 Static duration
Review: variables in functions
Member variables of classes
Member functions of classes
2 Namespaces
Background
Scope
Declarative regions
Potential Scope
Making namespaces
Acessing namespaces

Outline

1 Static duration
Review: variables in functions
Member variables of classes
Member functions of classes
2 Namespaces
Background
Scope
Declarative regions
Potential Scope
Making namespaces
Acessing namespaces

Review: Static variables in functions

- Recall, static duration variables are only created (and initialized) once, and then they are persisted beyond the scope of the function call, throughout the life of the program.
- Review the code from previously

Static members of a class

- Static members of a class are not associated with the objects of the class; they are independent objects with static storage duration or regular functions defined in namespace scope, only once in the program.
- The static keyword is only used with the declaration of a static member, inside the class definition, but not with the definition of that static member, for example:

```
// declaration (uses 'static')
class X { static int n; };

// definition (does not use 'static')
int X::n = 1;
```

Outline

1 Static duration
Review: variables in functions
Member variables of classes
Member functions of classes
2 Namespaces
Background
Scope
Declarative regions
Potential Scope
Making namespaces
Acessing namespaces

Static member variables in a class

- A static data member of a class is also known as a "class variable", because there is only one common variable for all the objects of that same class, sharing the same value: i.e., its value is not different from one object of this class to another.
- Static member variables (data members) are not initialied using constructor; they must be initialized explicitly outside the class (unless they're const integral or enum).
- Can be accessed either via *object*.*s_var* or via *classname* :: *s_var*, but the latter is preferred, since it is congruent with the design intentions (to be data for the whole class)
- Check out examples

Outline

1 Static duration
Review: variables in functions
Member variables of classes
Member functions of classes

2 Namespaces
Background
Scope
Declarative regions
Potential Scope
Making namespaces
Acessing namespaces

Static member functions of a class

- You just learned that static member variables are member variables that belong to the class rather than objects of the class.
- If the static member variables are public, we can access them directly using the class name and the scope resolution operator.
- But what if the static member variables are private?
- Use a static member function!

Static member functions of a class

- Static member functions are not associated with any object.
- Thus, when called, they have no this pointer.
- Second, static member functions can only access static member variables. They can not access non-static member variables. This is because non-static member variables must belong to a class object, and static member functions have no class object to work with!
- Because static member functions are not attached to a particular object, they can be called directly by using the class name and the scope resolution operator.
- Like static member variables, they can also be called through objects of the class type, though this is not recommended.
- Check out examples

Outline

1 Static duration
Review: variables in functions
Member variables of classes
Member functions of classes

2 Namespaces
Background
Scope
Declarative regions
Potential Scope
Making namespaces
Acessing namespaces

Outline

1 Static duration
Review: variables in functions
Member variables of classes
Member functions of classes

2 Namespaces
Background
Scope
Declarative regions
Potential Scope
Making namespaces
Acessing namespaces

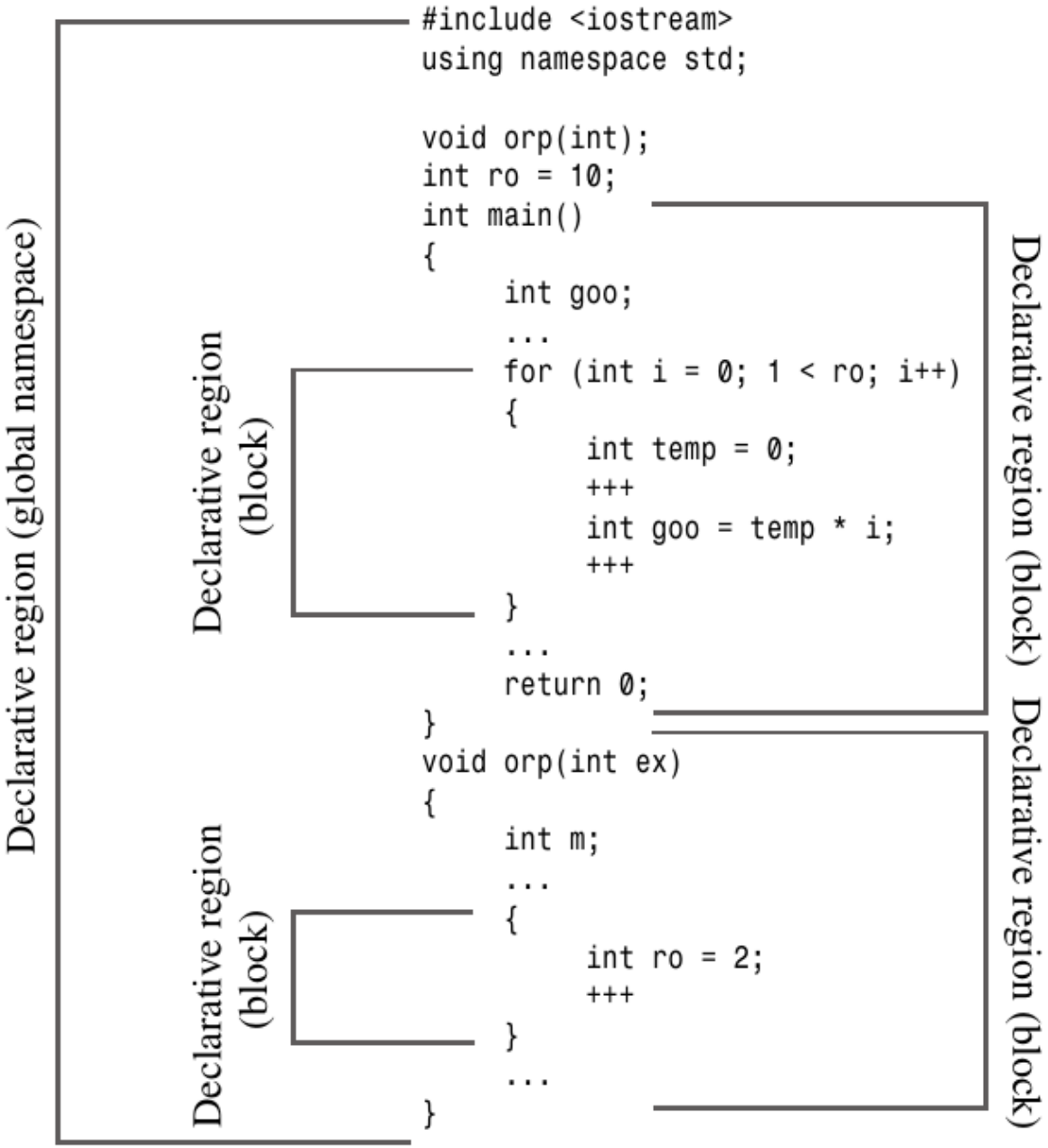
Scope


```
int foo;            // global variable

int some_function()
{
  int bar;          // local variable
  bar = 0;
}

int other_function()
{
  foo = 1;  // ok: foo is a global variable
  bar = 2;  // wrong: bar is not accessible
}
```

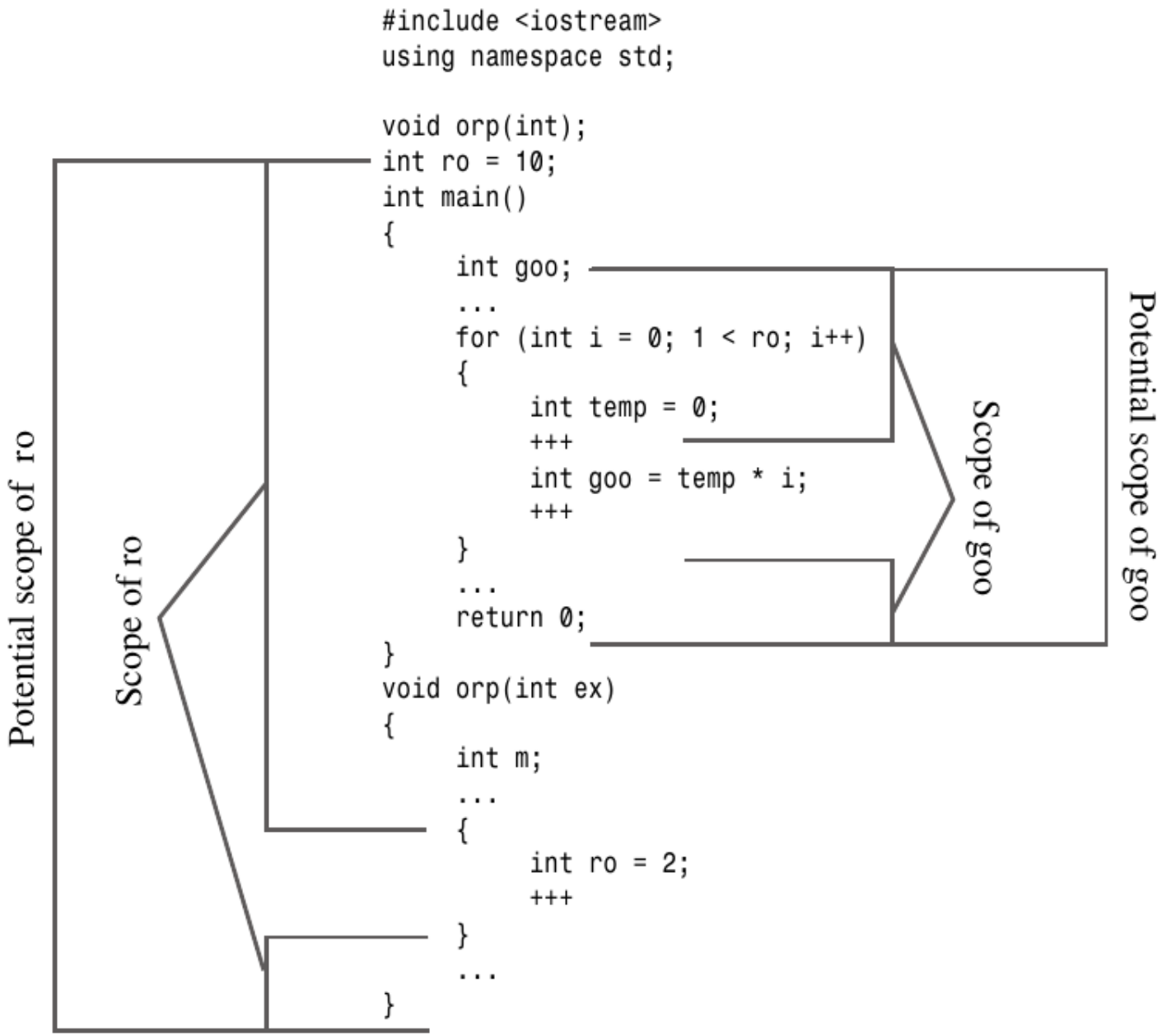
In each scope, a name can represent one entity


```
int some_function()
{
   int x;
   x = 0;
   double x; //name already used in scope
   x = 0.0;
}
```

Declarative regions



Potential scope of an object



Outline

1 Static duration
Review: variables in functions
Member variables of classes
Member functions of classes

2 Namespaces
Background
Scope
Declarative regions
Potential Scope
Making namespaces
Acessing namespaces

Making new namespaces

- Namespaces provide a method for preventing name conflicts in large projects.
- The namespace keyword allows you to create a new scope.
- Symbols declared inside a namespace block are placed in a named scope that prevents them from being mistaken for identically-named symbols in other scopes.
- Multiple namespace blocks with the same name are allowed, and these can be in multiple files. All declarations within those blocks are declared in the named scope.
- The namespace definition must be done at global scope, or nested inside another namespace.

General syntax


```
namespace namespace_name
{
    // code declarations
}

// code could be a variable or function
namespace_name::code;
```

Outline

1 Static duration
Review: variables in functions
Member variables of classes
Member functions of classes

2 Namespaces
Background
Scope
Declarative regions
Potential Scope
Making namespaces
Acessing namespaces

Three ways to access a namespace

1. Scope Resolution (preferred)
2. The using directive (not preferred)
3. The using declaration (preferred)

See code examples

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 library of functions or classes, place them in a namespace.
- Use the using directive only as a temporary means of converting old code to namespace usage.
- Don't use using directives in header files; doing so conceals which names are being made available. Also, the ordering of header files may affect behavior. If you use a using directive, place it after all the preprocessor #include directives.
- Preferentially import names by using the scope-resolution operator or a using declaration.
- Preferentially use local scope instead of global scope for using declarations.