

HIGH-LEVEL PROGRAMMING 2

Namespaces

by Prasanna Ghali

Objects, Variables, Types ...

2

- *Type* is set of values and set of operations on those values
- *Object* is region of memory that has a type
 - ▣ Every object has a type so we know what kind of information can be placed in that object
- *Variable* is a named object

```
int *pi = (int*) malloc(sizeof(int));
```

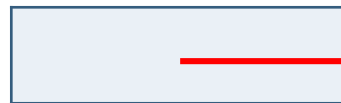
```
int age = 39;
```

int:

int*:

int:

pi:



(just an object)

age: 39

(both `age` and `pi` are objects that are also variables)

Several Kinds of Scopes

3

- There're several kinds of scopes that we use to control where our names are used:
 - ▣ *Global* or *file scope*: area of text outside any other scope
 - ▣ *Function scope*: area of text in function body including parameters
 - ▣ *Local scope*: between { . . . } braces of a block in a function
 - ▣ *Statement scope*: e.g., in a **for** statement

Main Purpose of Scope

4

- Main purpose of a scope is to keep names local, so they won't interfere with names declared elsewhere

```
void f(int x) // f is global; x is local to f
{
    int z = x+7; // z is local to f
}

int g(int x) // g is global; x is local to g
{
    int f = x+2; // f is local to g
    return 2*f;
}
```

Global scope

f:

x

z

g:

x

f

Scoping Rule

5

- Compiler will associate *closest declaration of a name* when references are made to variables, types, and functions with same name

Scope: Examples (1 / 4)

6

```
void f1(int param) { // scope of param starts here
    int a = 2;        // scope of a starts here
    int b = 10;       // scope of b starts here

    while (a < 10) {
        int x;        // scope of x starts here

        x = param * a++; // OK, param and a both in scope

        if (x == 5)
            b = 11;    // OK, b in scope
        }             // scope of x ends here

        x = 3;         // ERROR! x not in scope
    } // scope of a, b and param end here
```

Scope: Examples (2/4)

7

```
int x = 1;    // scope of global x starts here

void f2(int param) { // scope of param starts here
    int a = 2;      // scope of a starts here
    int param;      // ERROR! param already defined

    while (a < 10) {
        int x = 2;  // scope of second x starts here
        double a;   // OK. Different scope, different (second) a
        float param; // OK. Different scope, different param

        if (x == 5) {
            int x;    // OK. Different scope, different (third) x
            int param; // OK. Different scope, different param
        } // scope of third x and param ends here
    } // scope of second x, second a and param ends here
} // scope of first a and param ends here
```

Scope: Examples (3/4)

8

```
int a; // Memory for a is allocated before program starts
      // Value of a is 0 (global variable)

void f3(int param) {
    int b; // Memory for b is allocated when function starts
           // b is uninitialized (local variable)
    b = a * param; // OK, a and param in scope
}

int main() {
    int x; // Memory for x is allocated when main starts
           // x is uninitialized (because it is local variable)

    f3(x); // Memory for copy of x is allocated when f3 starts
           // Memory for copy of x is deallocated when f3 returns
}
// Memory for a is deallocated when program ends
```


Scope: Examples (4/4)

9

In C, we're declaring type `struct S`
while in C++, we're declaring type `S`

Can also be:

```
int f(struct S x) {  
    /*...*/  
}
```

```
struct S { // S is global  
    int x; // x is local to type S  
    int y; // y is local to type S  
};
```

```
int f(S x) { // f is global; x is local to f  
    int y = x.x+x.y; // y is local to f  
    return y;  
}
```

Namespaces

10

- C++ provides language feature, **namespace**, exclusively for expressing scoping
- Why and how?

Why Namespaces? (1 / 2)

11

- We use blocks to organize code within function
- We use structures to organize data and types into a type
- Both do two things for us:
 - ▣ They give us a name to refer to what we've defined
 - ▣ They allow us to define a number of “entities” without worrying that their names clash with other names in our program

Why Namespaces? (2/2)

12

- What we lack is a way to organize our functions, data, and types to prevent *name clashes* with names introduced by other programmers [in their source files] and by third-party C/C++ libraries [such as OpenGL and UDK]

Name Clashes (1 / 2)

13

- By default, in C/C++, names defined in global scope in a source file have *external linkage*
 - ▣ This means that compiler will export such names to linker
- If program contains multiple definitions with same name [having external linkage], C/C++ require these definitions to be similar
 - ▣ Otherwise, linker signals *name clash*

Name Clashes (2/2)

14

- Individual source files compile but linker fails!!!
- External linkage implies program consisting of many source files must have *one and only one definition of name defined in global scope*

```
// mine.cpp
int x = 10;

int main() {
    printf("mine: %d\n", x);
    extern void foo();
    foo();
}
```

```
// yours.cpp
int x = 20;

void foo() {
    printf("yours: %d\n", x);
}
```

What is a Namespace? (1 / 4)

15

- What we lack is a way to organize our functions, data, and types to prevent *name clashes* with names introduced by other programmers [in their source files] and by third-party C++ libraries [such as UDK]
- C++ mechanism for grouping of declarations is called a *namespace*
 - ▣ Namespace is a block that attaches an extra name – the namespace name – to every entity name that is declared within it

What is a Namespace? (2/4)

16

```
// in graphics.cpp ...  
struct Color { /* ... */ };  
struct Line { /* ... */ };  
struct Shape { /* ... */ };  
void draw(Shape *) { /* ... */ }  
void xform() { /* ... */ }  
// ...  
void gui() { /* ... */ }
```

This is how you define a namespace.

No semicolon is required after closing brace in namespace definition!!!

```
// in graphics.cpp ...  
namespace Graphics_Lib {  
    struct Color { /* ... */ };  
    struct Line { /* ... */ };  
    struct Shape { /* ... */ };  
    void draw(Shape *) { /* ... */ }  
    void xform() { /* ... */ }  
    // ...  
    void gui() { /* ... */ }  
}
```

By enclosing names **Color**, **Line**, ... in namespace **Graphics_Lib** in my source file, name clashes are prevented if you use the same names in your source files

What is a Namespace? (3/4)

17

Full name of each entity is the namespace name followed by scope resolution operator, `::`, followed by the entity name, as in:

`Graphics_Lib::gui` means
“name `gui` in namespace scope
`Graphics_Lib`”

```
// in graphics.cpp ...
namespace Graphics_Lib {
    struct Color { /* ... */ };
    struct Line { /* ... */ };
    struct Shape { /* ... */ };
    void draw(Shape *) { /* ... */ }
    void xform() { /* ... */ }
    // ...
    void gui() { /* ... */ }
}
```

What is a Namespace? (4/4)

18

```
// in graphics.cpp ...
namespace Graphics_Lib {
    struct Color { /* ... */ };
    struct Line { /* ... */ };
    struct Shape { /* ... */ };
    void draw(Shape *) { /* ... */ }
    void xform() { /* ... */ }
    // ...
    void gui() { /* ... */ }
}
```

```
// in yours-graphics.cpp ...
namespace Better_Graphics_Lib {
    struct Color { /* ... */ };
    struct Line { /* ... */ };
    struct Shape { /* ... */ };
    void draw(Shape *) { /* ... */ }
    void xform() { /* ... */ }
    // ...
    void gui() { /* ... */ }
}
```

Different namespaces can contain entities with the same name, but the entities are differentiated because they're *qualified* by different namespace names, as in:

Graphics_Lib::Shape is a different entity compared to Better_Graphics_Lib::Shape

Global Namespace (1 / 3)

19

- *Global namespace* applies by default if a namespace hasn't been defined

```
#include <iostream>

int global_int{10};

int main() {
    std::cout << global_int << '\n';
}
```

All names within global namespace are just as you declare them, without a namespace being attached

Global Namespace (2/3)

20

- To explicitly access names defined in global namespace, you use scope resolution operator without left operand

```
#include <iostream>

int global_int{10};

int main() {
    std::cout << ::global_int << '\n';
}
```

Global Namespace (3/3)

21

- Explicit use of scope resolution operator to access names in global namespace only really required if there is a more local declaration with same name that *hides the global name*

```
#include <math.h>

double zip_a_dee(int pow) {
    return ::pow(3.141592, pow);
}
```

Recall that `pow` is a function declared in C standard library header `math.h`!!!

Global Namespace and C Standard Library (1 / 2)

22

- For compatibility with C standard library, C++ standard library provides headers just as in C source files

Just as in C code, C standard library names are also in global scope in C++ code if they're exposed by including header files with .h suffix

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *pi { (int*) malloc(sizeof(int)) };
    *pi = 39;
    ::printf("*pi: %d\n", *pi);
}
```


Global Namespace and C Standard Library (2/2)

23

- For compatibility with C standard library, C++ standard library provides headers just as in C source files

Don't do this!!!

Will be deprecated in future C++ versions!!!



```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *pi { (int*) malloc(sizeof(int)) };
    *pi = 39;
    printf("*pi: %d\n", *pi);
}
```

Namespace `std` and C++ Standard Library

24

Standard header declares global variables that control reading from and writing to standard streams `stdout`, `stdin`, and `stderr`

Namespaces are C++ mechanisms that introduce new scopes to avoid conflicts between names in large programs.

`std` is namespace for virtually all names in C++ standard library

```
#include <iostream>

int main() {
    std::cout << "Hello World\n";
}
```

`::` is *scope resolution operator*.
`std::cout` means “name `cout` in namespace scope `std`”

Global variable of type `std::ostream` is instantiated at program startup and its purpose is to write characters to standard stream `stdout`

Namespace `std` and C Standard Library

25

- Facilities of C standard library in header *name.h* provided in C++ standard library header *cname*:

```
#include <cstdio>
#include <cstdlib>
```

Do this!!!

Names in these headers
are within namespace `std`

```
int main() {
    int *pi { (int*) std::malloc(sizeof(int)) };
    *pi = 39;
    std::printf("*pi: %d\n", *pi);
}
```

Defining a Namespace [Revisited]

26

- Namespace block doesn't have to be contiguous

```
namespace Math {  
    double const sqrt2{1.4142135};  
}  
  
namespace Graphic_Lib {  
    struct Color { /* ... */ };  
}  
  
namespace Math {  
    double square(double x) {  
        return x*x;  
    }  
}
```

You can extend a namespace scope by adding a second namespace block with same name

Namespaces can be spread across files too!!!

Functions and Namespaces

27

- For function to exist within a namespace, it is sufficient for function declaration to appear in a namespace block

```
namespace Math {  
    double const sqrt2{1.4142135};  
    double square(double x);  
}
```

We've two options for defining the function

```
// option 1: enclose in  
// namespace block  
namespace Math {  
    double square(double x) {  
        return x*x;  
    }  
}
```

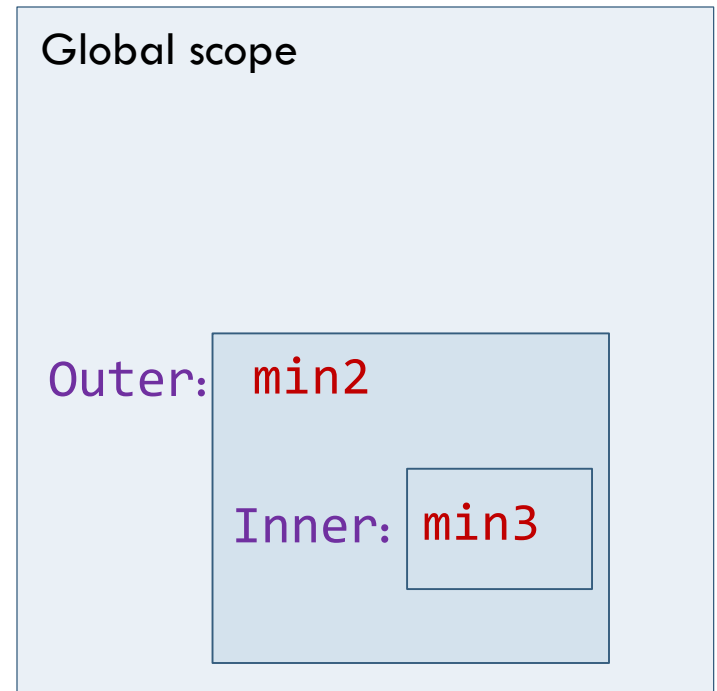
```
// option 2: use qualified name  
// of function  
double Math::square(double x) {  
    return x*x;  
}
```

Nested Namespaces (1/2)

28

- You can define one namespace inside another

```
namespace Outer {  
    int min2(int x, int y) {  
        return x < y ? x : y;  
    }  
  
    // option 1: explicit nesting  
    namespace Inner {  
        int min3(int x, int y, int z) {  
            return min2(x, min2(y, z));  
        }  
    }  
}
```

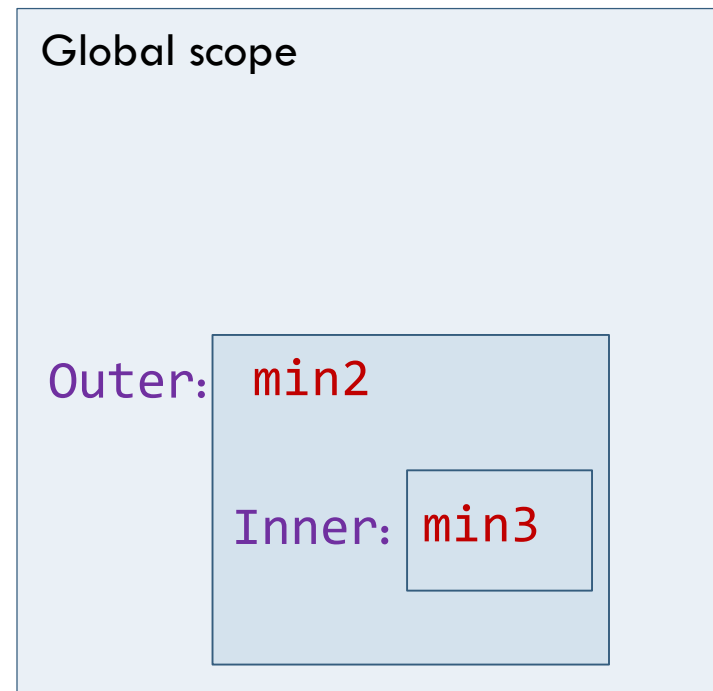


Nested Namespaces (2/2)

29

- You can add something to nested namespace directly using following compact system:

```
namespace Outer {  
    int min2(int x, int y) {  
        return x < y ? x : y;  
    }  
}  
  
// option 2: direct nesting  
namespace Outer::Inner {  
    int min3(int x, int y, int z) {  
        return min2(x, min2(y, z));  
    }  
}
```



Anonymous [or Unnamed] Namespaces (1 / 3)

30

- Anonymous namespace looks like this:

```
namespace {  
    int global_int{10};  
  
    int update(int i) {  
        return global_int = i;  
    }  
  
    int add(int i) {  
        return i+global_int;  
    }  
}
```

Why use anonymous namespaces?

Anonymous [or Unnamed] Namespaces (2/3)

31

- One good reason is that variables and functions declared in anonymous namespaces in a source file have *internal linkage*!!!

```
namespace {  
    int global_int{10};  
  
    int update(int i) {  
        return global_int = i;  
    }  
  
    int add(int i) {  
        return i+global_int;  
    }  
}
```

```
// equivalent to declarations  
// in anonymous namespace  
static int global_int{10};  
  
static int update(int i) {  
    return global_int = i;  
}  
  
static int add(int i) {  
    return i+global_int;  
}
```

Anonymous [or Unnamed] Namespaces (3/3)

32

- Second good reason is that anonymous namespaces can be used to declare types that can only be referenced from one source file

```
namespace {  
    // type S is "private" to this source file  
    struct S { int x, y; };  
  
    // function print is also "private" to this source file  
    void print(S const *ps) {  
        std::cout << ps->x << ',' << ps->y << '\n';  
    }  
}
```


Namespace Aliases (1 / 3)

33

- Long namespace names may be unduly cumbersome to use

```
namespace Programming {  
    namespace AdvancedProgramming {  
        int foo{11}, bar{12};  
        int f1(int x) { return x / 2; }  
    }  
  
    namespace IntroductoryProgramming {  
        int foo{21}, bar{22};  
        int Div2(int x) {return x / 2; }  
    }  
}  
  
int main() {  
    std::cout << Programming::AdvancedProgramming::foo;  
    std::cout << Programming::IntroductoryProgramming::Div2(8);  
}
```

Namespace Aliases (2/3)

34

- You can create *aliases* to shorten namespace names

```
namespace Programming {
    namespace AdvancedProgramming {
        int foo{11}, bar{12};
        int f1(int x) { return x / 2; }
    }

    namespace IntroductoryProgramming {
        int foo{21}, bar{22};
        int Div2(int x) {return x / 2; }
    }
}

namespace PAP = Programming::AdvancedProgramming;
namespace PIP = Programming::IntroductoryProgramming;

int main() {
    std::cout << PAP::foo;
    std::cout << PIP::Div2(8);
}
```

using Declaration

35

- Allows you to use a *specific name* without its namespace qualification

```
namespace Stuff {  
    int foo = 11; // Stuff::foo  
    int bar = 12; // Stuff::bar  
}  
  
using Stuff::foo;  
  
void f1() {  
    Stuff::foo = 21; // ok: using qualified name  
    foo = 22;       // ok: namespace not required  
    using Stuff::bar; // make bar available in this scope only  
    bar = 30;       // ok  
    //int bar = 5;   // error: redeclaring name bar  
}
```

C Standard Library and HLP2

Assessments

36

- Unless explicitly specified by assessment specification, expect zero grade for assessments that rely on C standard library for I/O, dynamic memory allocation/deallocation, ...