Object Oriented Programming with C++

3. C++ namespaces, operators, and expressions

By: Prof. Pandav Patel

Second Semester, 2020-21
Computer Engineering Department
Dharmsinh Desai University

C++ namespaces

- New concept introduced in C++
- Declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it
- Used to organize code into logical groups and to prevent name collisions (e.g. when code includes multiple libraries)
- Scope resolution operator (::) can be used to access variables in other scopes
- ***using*** keyword can be used to introduce specific members of a namespace in to the current scope or all members of the namespace in to the scope of common parent
- Namespaces can be nested

C++ namespaces

```
#include<iostream>
using namespace std;
namespace nspace
{
	int x = 11;
	void fun()
	{
		cout << "Hello!" << endl;
	}
}
int main()
{
	//fun(); // Invalid
	nspace::fun();
	//cout << x << endl; // Invalid
	cout << nspace::x << endl;
	return 0;
}
```

C++ namespaces

```
#include<iostream>
using namespace std;
namespace nspace
{
	int x = 11;
	void fun()
	{
		cout << "Hello!" << endl;
	}
}
using nspace::fun; //using declaration
int main()
{
	fun();
	nspace::fun();
	//cout << x << endl; // Invalid
	cout << nspace::x << endl;
	return 0;
}
```

C++ namespaces

```
#include<iostream>
using namespace std;
namespace nspace
{
	int x = 11;
	void fun()
	{
		cout << "Hello!" << endl;
	}
}
using namespace nspace; //using directive
int main()
{
	fun();
	nspace::fun();
	cout << x << endl;
	cout << nspace::x << endl;
	return 0;
}
```

C++ namespaces

```
#include<iostream>
using namespace std;
int x = 10;
namespace nspace
{
	int x = 20;
	void fun()
	{
		int x = 30;
		cout << x << " " << nspace::x << " " << ::x << endl;
	}
}
int main()
{
	nspace::fun();
	return 0;
}
```

C++ namespaces

- A namespace can be split into multiple blocks and those blocks can be in the same file or separate files.
- One file can contain blocks of multiple namespaces

1.cpp

```
namespace abc
{
	...
}

namespace xyz
{
	...
}

namespace abc
{
	...
}
```

2.cpp

```
namespace xyz
{
	...
}

namespace abc
{
	...
}

namespace xyz
{
	...
}
```

C++ namespaces (using declaration)

```
#include<iostream>
int x = 1;
namespace nspace
{
	int x = 2;
	int y = 3;
}
int main()
{
	int y = 4;
	//error: ‘y’ is already declared in this scope
	//using nspace::y;
	using nspace::x;
	std::cout << x << " " << y;
	int fun();
	fun();
	return 0;
}
int fun()
{
	std::cout << x;
}
```

- Here **nspace::x** has been introduced within main function scope

C++ namespaces (using declaration)

```
#include<iostream>

namespace test {
    int x = 7;
}

namespace test2 {
    using test::x;
}

int main() {
    std::cout << test2::x;
    return 0;
}
```

- **Output**
- 7

C++ namespaces (using directive)

```
#include<iostream>

int x = 1;

namespace nspace
{
    int x = 2;
    int y = 3;
    int z = 4;
}
```

- Qualified name lookup for ::x, ::y and ::z, starts from global scope, and only ::x is considered as part of global scope. naspace::y and naspace::z are not considered part of global scope for qualified name lookup.

```
int main()
{
    int y = 5;
    using namespace nspace;
    //error: reference to 'x' is ambiguous
    //std::cout << " " << x;
    std::cout << " " << ::x;
    std::cout << " " << y;
    //error: '::y' has not been declared
    //std::cout << " " << ::y;
    std::cout << " " << z;
    //error: '::z' has not been declared
    //std::cout << " " << ::z;
    int fun();
    fun();
    return 0;
}
void fun()
{
    std::cout << " " <<  x;
}
```

- **nspace::x** is qualified name of **x**.
- **Unqualified name** is a name that does not appear to the right of a scope resolution operator ::
- x is unqualified name of variable x.

- For **unqualified name lookup** form **main** function (**after using directive statement**) it is as if all members of the namespace nspace are part of **global scope**
- Produces error **only if** we **try to access x** within main function (**after using directive statement**)
- Unqualified name lookup for x starts from main (local scope), but x is not present in main hence it checks global scope where it find more than one declaration of x hence the error
- Unqualified name lookup for y starts from main (local scope), and y is present in main hence search stops there and it uses local y from main
- Unqualified name lookup for z starts from main (local scope), but z is not present in main hence it checks global scope where it find one declaration of z.

C++ namespaces (using directive)

```
#include<iostream>

int x = 1;

namespace nspace
{
    int x = 2;
    int y = 3;
    int z = 4;
}
```

```
using namespace nspace;
int main()
{
    int y = 5;
    //error: reference to ‘x’ is ambiguous
    //std::cout << " " << x;
    std::cout << " " << ::x;
    std::cout << " " << y;
    std::cout << " " << ::y;
    std::cout << " " << z;
    std::cout << " " << ::z;
    int fun();
    fun();
    return 0;
}
void fun()
{
    //error: reference to ‘x’ is ambiguous
    //std::cout << " " <<  x;
}
```

- Avoid **using namespace std;** as it pollutes the global scope for unqualified name lookup.

- Qualified name lookup for ::x, ::y and ::z, starts from global scope, and only ::x is considered as part of global scope. naspace::y and naspace::z are not considered part of global scope for qualified name lookup.
- In this case using directive statement is present in global scope. **So when y and z are not found in global scope, search continues within namespaces for which using directive is present in global scope (nspace in this case).** And hence search for ::y and ::z ends with nspace::y and nspace::z respectively.

- As using directive statement is present in global scope now, for **unqualified name lookup after using directive statement,** it is as if all members of the namespace nspace are part of **global scope**
- Hence unqualified name lookup for **x** from function **fun** results in ambiguity.

C++ namespaces (using directive)

```
#include<iostream>

namespace test {
    int x = 7;
}

namespace test2 {
    using namespace test;
}

int main() {
    std::cout << test2::x;
    return 0;
}
```

- **Output**
- 7

- Qualified name lookup for test2::x starts from test2. So when x is not found in test2, search continues within namespaces for which using directive is present in test2 scope (test namespace in this case). And hence search for test2::x ends with test::x.

C++ namespaces (using directive)

```
#include<iostream>
int x = 1;
namespace test {
    namespace test2 {
        int x = 2;
    }
    namespace test3 {
        using namespace test2;
        void fun() {
            std::cout << x;
            //error: ‘x’ is not a member of ‘test’
            //std::cout << test::x;
            std::cout << " " << ::x;
        }
    }
}
int main() {
    test::test3::fun();
    return 0;
}
```

- **Output**
- 2 1

- Qualified name lookup for test::x starts from test namespace. So when x is not found in test namespace, search continues within namespaces for which using directive is present in test namespace (there is no using directive directly within test). Hence search stops there without finding x, and hence compiler generates an error.

- Qualified name lookup for ::x starts directly from global scope and x is present in global scope.

- For unqualified lookup for x, it searches in local scope (fun function scope) first. Variable x is not present in fun function and hence it will look into parent scope of fun scope (test3). x is not present in test3. Hence search continues to its parent (test). Variable x is found in test. Variable x is considered part of test (for unqualified name lookup) because of using directive in test3.

C++ Operators

- All C language operators are valid in C++
- C++ introduces some new operators
  - Insertion operator (<<) and extraction operator (>>)
  - Scope resolution operator (::)
  - Member dereferencing operators (::*, ->*, .*)
  - Memory management operators (new and delete)
  - Type cast operator

  - Manipulators (endl, setw etc)

C++ Operators

- Memory management operators (new and delete)
  - Like C language, **malloc, calloc, realloc** and **free** *functions* will still work
  - New operators **new** and **delete** are added. Keyword **new** is used to allocate memory while **delete** is used for deallocating memory allocated using **new**.
  - **new** and **delete** are *operators* (and *keywords*). A.k.a Free store operators
  - Like **malloc, calloc** and **realloc**, lifetime of memory allocated using **new** is controlled by programmer
  - No inbuilt feature for garbage collection like other OOP languages

  - You can use **malloc** and **new** in the same program. But you cannot allocate an object with **malloc** and free it using **delete**. Nor can you allocate with **new** and delete with **free** or use **realloc** on an array allocated by new.
  - Use **std::vector** if you ever need to **realloc** (**realloc** would still work for memory allocaed using **malloc** and **calloc,** but it is discouraged in favor of **std::vector**)

C++ Operators

- Memory management operators (new and delete)
  - pointer-variable = new data-type(value);
    - data-type could be fundamental or user-defined
    - pointer-variable is pointer to data-type
    - e.g. **int *p1; p1 = new int;**
    - Safe to assume that it is initialized with garbage by default
    - e.g. **int *p2 = new int; *p2 = 10;**
    - e.g. **float *p3 = new float(25.5);**

  - delete pointer-variable;
    - pointer-variable must be pointer returned by new (which is not already deleted)
    - e.g. **delete p1;** //Frees memory, does not delete pointer itself

C++ Operators

- Memory management operators (new and delete)
  - Memory can be allocated using **new** for array
    - pointer-variable = new data-type[10];
    - e.g. **int *p4 = new int[10];**
    - e.g. **int *p5 = new int[10]();** //Initialize all elements with 0 c++03
    - e.g. **int *p6 = new int[10]{1, 2, 3};** //c++11
    - In case of multi-dimentional arrays, all sizes must be supplied
      - e.g. **int (*p7)[5] = new int[4][5];**
      - e.g. **int (*p8)[5] = new int[m][5];**
        - First dimention can be variable, others must be constant.
        - Why??
  - delete [size] pointer-variable;
    - e.g. **delete [] p4;**
    - e.g. **delete [] p7;**

C++ Operators

- Memory management operators (new and delete)
  - If call to **new** fails it will throw exception and programm will terminate unless you handle that exception by catching it (will learn later)
    - It would throw **bad_alloc** exception
  - If call to **delete** fails then also program will terminate like **free**
    - It would not throw exception
  - It is good practice to free the memory when it is no longer required
  - If you do not free the memory explicitly, it will be freed when program execution ends
  - Advantages of new over malloc
    - Automatically computes size
    - Returns correct pointer type (No explicit type cast needed)
    - Possible to initialize value while allocating memory
    - **new** and **delete** operators could be overloaded

Memory management operators (new and delete)

```
#include<iostream>

using std::cout, std::endl;

int *fun() {
    int *num_ptr = new int(7);
    cout << *num_ptr << " " << num_ptr << endl;
    return num_ptr;
}

int main() {
    int *ptr = fun();
    cout << *ptr << " " << ptr << endl;
    delete ptr;
    //ptr is a dangling pointer from here on
    //using ptr will result in undefined behaviour
    cout << *ptr << " " << ptr << endl;
    return 0;
}
```

```
#include<iostream>

using std::cout, std::endl;

int &fun() {
    int &num_ref = *(new int(7));
    cout << num_ref << " " << &num_ref << endl;
    return num_ref;
}

int main() {
    int &ref = fun();
    cout << ref << " " << &ref << endl;
    delete &ref;
    //ref is a dangling reference from here on
    //using ref will result in undefined behaviour
    cout << ref << " " << &ref << endl;
    return 0;
}
```

C++ Operators

- Type cast operator
  - (type-name) expression; //C style, still valid in C++
    - e.g. **int i = (int)f; int i = (int)5.3;**
  - type-name(expression); //C++ style
    - e.g. **int i = int(f); int i = int(5.3);**
    - Can only be used if type-name is following the rules of identifier
      - p = int * (q); // Invalid
      - typedef int * int_pt; p = int_pt(q); // Valid

C++ Operators - Manipulators

- Manipulators
  - If a pointer to function (which returns **ostream** reference and takes **ostream** reference as first argument) is given as the second argument to **<<** , the function pointed to is called. For example, **cout << pf** means **pf(cout)** . Such a function is called a manipulator.
  - **>>** needs - function which returns **istream** reference and takes **istream** reference as first argument – to work as manipulator for input streams.
  - manipulator is a function that can be invoked by **<<** or **>>**
  - Manipulators are used to format data
    - Can be used to format input and output streams. But frequently used to format output streams.
    - **Iomanip** contains declaration for many such manipulators
      - These manipulators internally call member functions of **ios** class

C++ Operators - Manipulators

- Manipulators (Only few important ones) **Explore more here**

Manipulator	Meaning
endl	Insert newline and flush stream
setw(int w)	Set field width to w. It only applies to next value to be inserted, then it is reset to default (0)
setfill(int c)	Set the fill character to c. Default is space.
left	Append fill characters at the end
right	Insert fill characters at the beginning
setprecision(int d)	Set the floating point precision to d.
fixed	Floating-point values are written using fixed-point notation: the value is represented with exactly as many digits in the decimal part as specified by the precision field

C++ Operators – Manipulators

```
#include<iostream>
#include<iomanip>
int main()
{
	std::cout << std::setw(10) << "Hello" << std::endl;
	std::cout << "Hello" << std::endl;
	std::cout << std::setw(2) << "Hello" << std::endl << std::endl;

	std::cout << std::left << std::setfill('*');
	std::cout << std::setw(9) << "Hello" << std::endl;
	std::cout << std::setw(7) << "Hello" << std::endl << std::endl;

	std::cout << std::setprecision(2) << 111.11111 << std::endl;
	std::cout << 1.11111 << std::endl;
	std::cout << std::setprecision(10) << 111.11111 << std::endl;
	std::cout << 1.11111 << std::endl << std::endl;

	std::cout << std::fixed;
	std::cout << std::setprecision(2) << 111.11111 << std::endl;
	std::cout << 1.11111 << std::endl;
	std::cout << std::setprecision(10) << 111.11111 << std::endl;
	std::cout << 1.11111 << std::endl << std::endl;

	return 0;
}
```

```
     Hello
Hello
Hello

Hello****
Hello**

1.1e+02
1.1
111.11111
1.11111

111.11
1.11
111.1111100000
1.1111100000
```

C++ Operators - Manipulators

```
#include<iostream>

std::ostream & have_fun(std::ostream &output)
{
	output << std::endl << "Have Fun";
	return output;
}

int main()
{
	std::cout << "Hello there,";
	std::cout << have_fun << ", you guys!!" << std::endl;

	std::endl(have_fun(std::cout) << ", you guys!!");

	return 0;
}
```

```
Hello there,
Have Fun, you guys!!

Have Fun, you guys!!
```

C++ Expressions and implicit conversions

- Avoid mix of **signed** and **unsigned** numbers in an expression
- Explicitly type-casting is good practice to avoid confusion

- During evaluation of an expression
  - All **char** and **short** - variables and constants - are converted to **int** first
  - Then smaller type is converted to wider type before applying operator
    - int, long, float, double, long double
    - e.g. ‘A’ + 3 + 6.5 // result would be double as 6.5 is double

C++ Expressions and implicit conversions

- Operator precedence and associativity does not guarantee order of evaluation

```
#include<iostream>
int i = 2;
int fun()
{
	i++;
	return i;
}
int main()
{
	int n = fun() + fun() * fun();
	std::cout << n << std::endl;
	return 0;
}
// Prints 23 for me on g++, could be something else in your case
```

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right
2	`a++ a--`	Suffix/postfix increment and decrement	
	`type() type{}`	Functional cast	
	`a()`	Function call	
	`a[]`	Subscript	
	`. ->`	Member access	
3	`++a --a`	Prefix increment and decrement	Right-to-left
	`+a -a`	Unary plus and minus	
	`! ~`	Logical NOT and bitwise NOT	
	`(type)`	C-style cast	
	`*a`	Indirection (dereference)	
	`&a`	Address-of	
	`sizeof`	Size-of[note 1]	
	`new new[]`	Dynamic memory allocation	
	`delete delete[]`	Dynamic memory deallocation	
4	`.* ->*`	Pointer-to-member	Left-to-right
5	`a*b a/b a%b`	Multiplication, division, and remainder	
6	`a+b a-b`	Addition and subtraction	
7	`<< >>`	Bitwise left shift and right shift	
8	`<=>`	Three-way comparison operator (since C++20)	
9	`< <=`	For relational operators < and ≤ respectively	
	`> >=`	For relational operators > and ≥ respectively	
10	`== !=`	For relational operators = and ≠ respectively	
11	`&`	Bitwise AND	
12	`^`	Bitwise XOR (exclusive or)	
13	`|`	Bitwise OR (inclusive or)	
14	`&&`	Logical AND	
15	`||`	Logical OR	
16	`a?b:c`	Ternary conditional[note 2]	Right-to-left
	`throw`	throw operator	
	`=`	Direct assignment (provided by default for C++ classes)	
	`+= -=`	Compound assignment by sum and difference	
	`*= /= %=`	Compound assignment by product, quotient, and remainder	
	`<<= >>=`	Compound assignment by bitwise left shift and right shift	
	`&= ^= |=`	Compound assignment by bitwise AND, XOR, and OR	
17	`,`	Comma	Left-to-right

C++ Operator Overloading

- Example of operator overloading is **<<** operator
  - Inserts variables or constants on the right to ostream on left
    - It effectively handles all different types of values on RHS
  - Bitwise left shift if integer on left
  - Calls function on the right in case of manipulators

- Developer can overload operators to give them a special meaning
  - For example, if you have created a structure, then you can overload operator **+** to add two variables of that structure by simply writing statement (**s1 + s2**), where **s1** and **s2** are structure variables
  - Member-access operators (**.** and **.***), conditional operator (**?:**), scope resolution operator (**::**) and **sizeof** operator can not be overloaded

Interesting reads

- Namespace must be existing before it is used with using directive
  - https://stackoverflow.com/questions/62876676/c-using-namespace-directive-for-non-existing-namespace#62876701
  - https://gcc.gnu.org/bugzilla/show_bug.cgi?id=29556
  - https://stackoverflow.com/questions/6841130/ordering-of-using-namespace-std-and-includes
- Qualified name lookup Vs Unqualified name lookup
  - https://en.cppreference.com/w/cpp/language/qualified_lookup
  - https://en.cppreference.com/w/cpp/language/unqualified_lookup
- More about namespaces
  - https://en.cppreference.com/w/cpp/language/namespace
- Sequence point
  - https://en.wikipedia.org/wiki/Sequence_point#Sequence_points_in_C_and_C++