27-28-29-31

Tuesday, March 30, 2021 7:22 PM

Chapter 27: The static Specifier

What is it ?

- The object will have a static storage duration. The memory space for static objects is allocated when the program starts and deallocated when program ends.
- Only one instance of static object exists in the program.
- If a local variable marked as static, the space for it is allocated the first time the program ontrol encounters its definition and deallocated when the program exits.

Syntax

```cpp
#include <iostream>
void myfunction()
{
static int x = 0;
x++;
std::cout << x << '\n';
}
int main()
{
myfunction(); // x == 1
myfunction(); // x == 2
myfunction(); // x == 3
}
```

```
1
2
3

Process returned 0 (0x0)   execution time : 0.035 s
Press any key to continue.
```

This variable is initialized the first time the program encounters this function. The value of this variable is preserved across function calls. The last changes we made to it *stays*. It will not get initialized to 0 for every function call, only the first time.

```cpp
#include <iostream>
class MyClass
{
public:
static int x; // declare a static data member
};
int MyClass::x = 123; // define a static data member
int main()
{
MyClass::x = 456; // access a static data member
std::cout << "Static data member value is: " << MyClass::x;
}
```

```
Static data member value is: 456
Process returned 0 (0x0)   execution time : 0.039 s
Press any key to continue.
```

We can define static class member fields. Static class members are not part of the object. They live independently of an object of a class. We declare a static data member inside the class and define it outside the class only once

- Here we declared a static data member inside a class. Then we defined it outside the class.
- When defining a static member outside the class, we do not need to use the static specifier. Then, we access the data member by using the MyClass::data_member_name notation.

```cpp
#include <iostream>
class MyClass
{
public:
static void myfunction(); // declare a static member function
};
// define a static member function
void MyClass::myfunction()
{
std::cout << "Hello World from a static member function.";
}
int main()
{
MyClass::myfunction(); // call a static member function
}
```

```
Hello World from a static member function.
Process returned 0 (0x0)   execution time : 0.032 s
Press any key to continue.
```

To define a static member function, we prepend the function declaration with the *static* keyword. The function definition outside the class does not use the *static* keyword

Chapter 28 : Templates

What is it?

- Templates are mechanisms to support the so-called generic programming- means we can define a funcction or a class without worrying a bout what types it accepts
- We define those functions and classes using some generic type
- When instantiating them, we use a conrete type, **so we can use templates when we want to denie a class or a function that can accept almost any type**

Syntax:

`template <typename T>` `// the rest of our function or class code`	`template <class T>` `// the rest of our function or class code`	T here stands for a type name. Which type? Well, any type. Here T means, *for all types T*

A function that can accept any type of argument:

```cpp
#include <iostream>
template <typename T>
void myfunction(T param)
{
std::cout << "The value of a parameter is: " << param;
}
int main()
{ }
```

To instantiate a function template, we call a function by supplying a specific type name, surrounded by angle brackets:

```cpp
#include <iostream>
template <typename T>
void myfunction(T param)
{
std::cout << "The value of a parameter is: " << param;
}
int main()
{
myfunction<int>(123);
myfunction<double>(123.456);
myfunction<char>('A');
}
```

```
The value of a parameter is: 123The value of a parameter is: 123.456The value of a parameter is: A
Process returned 0 (0x0)   execution time : 0.040 s
Press any key to continue.
```

-Obviously we can put any type of parameter inside that function

```cpp
#include <iostream>
template <typename T, typename U>
void myfunction(T t, U u)
{
std::cout << "The first parameter is: " << t << '\n';
std::cout << "The second parameter is: " << u << '\n';
}
int main()
{
int x = 123;
double d = 456.789;
myfunction<int, double>(x, d);
}
```

```
The first parameter is: 123
The second parameter is: 456.789
```

Templates can have more than one parameter. We simply list the template parameters and separate them using a comma

To define a class-template:

```cpp
#include <iostream>
template <typename T>
class MyClass {
private:
T x;
public:
MyClass(T xx)
:x{ xx }
{ }
T getvalue()
{
return x;
}
};
int main()
{
MyClass<int> o{ 123 };
std::cout << "The value of x is: " << o.getvalue() << '\n';
MyClass<double> o2{ 456.789 };
std::cout << "The value of x is: " << o2.getvalue() << '\n';
}
```

```
The value of x is: 123
The value of x is: 456.789

Process returned 0 (0x0)   execution time : 0.033 s
Press any key to continue.
```

Here, we defined a simple class template. The class accepts types T. We use those types wherever we find appropriate in our class. In our main function, we instantiate those classes with concrete types int and double. Instead of having to write the same code for two or more different types, we simply use a template.

```cpp
#include <iostream>
template <typename T>
class MyClass {
private:
T x;
public:
MyClass(T xx);
};
template <typename T>
MyClass<T>::MyClass(T xx)
: x{xx}
{
std::cout << "Constructor invoked. The value of x is: " << x << '\n';
}
int main()
{
MyClass<int> o{ 123 };
MyClass<double> o2{ 456.789 };
}
```

To define a class template member functions outside the class, we need to make them templates themselves by prepending the member function definition with the appropriate template declaration. In such definitions, a class name must be called with a template argument.

Template specialization

If we want our template to behave differently for a specific type, we provide the socalled template specialization. In case the argument is of a certain type, we sometimes want a different code.

Syntax:

```
template <>
// the rest of our code
To specialize our template function for type int, we write:
#include <iostream>
template <typename T>
void myfunction(T arg)
{
std::cout << "The value of an argument is: " << arg << '\n';
}
template <>
// the rest of our code
void myfunction(int arg)
{
std::cout << "This is a specialization int. The value is: " << arg <<
'\n';
}
int main()
{
```

```
myfunction<char>('A');
myfunction<double>(345.678);
myfunction<int>(123); // invokes specialization
}
```

Chapter 29: Enumerations

Enumeration, or *enum* for short, is a type whose values are user-defined named constants called *enumerators*

Unscoped enums	`enum MyEnum { myfirstvalue, mysecondvalue, mythirdvalue }; int main() { MyEnum myenum = myfirstvalue; myenum = mysecondvalue; // we can change the value of our enum object }`
we can change the underlying type	`enum MyEnum { myfirstvalue = 10, mysecondvalue, mythirdvalue };`

- These unscoped enums have their enumerators leak into an outside scope, the scope in which the enum type itself is defined.
- Old enums are best avoided. Prefer scoped enums to these old-school, unscoped enums.
- Scoped enums do not leak their enumerators into an outer scope and are not implicitly convertible to other types.

`enum class MyEnum { myfirstvalue, mysecondvalue, mythirdvalue }; int main() { MyEnum myenum = MyEnum::myfirstvalue; }`	To access an enumerator value, we prepend the enumerator with the enum name and a scope resolution operator :: such as MyEnum::myfirstvalue, MyEnum:: mysecondvalue, etc.
`enum class MyCharEnum : char { myfirstvalue, mysecondvalue, mythirdvalue };`	With these enums, the enumerator names are defined only within the enum internal scope and implicitly convert to underlying types. We can specify the underlying type for scoped enum:
`enum class MyEnum { myfirstvalue = 15, mysecondvalue, mythirdvalue = 30 };`	We can also change the initial underlying values of enumerators by specifying the value

Chapter 31: Organizing code

1. Header and Source Files

- Header files are source code files where we usally put various declarations.
- has .h or .hpp extension
- Source files are where we can store our definitions and the main program (.cpp extension)
- To include a standard library header, we use the #include statement followed by a header name without an extension, enclosed in bracket:

Example:

```
#include <iostream>
#include <string>
```

To include user-defined header files, we use the #include statement, followed by a full header name with extension enclosed in double-quotes. Example:

```
#include "myheader.h"
#include "otherheader.h"
```

2. Header Guards

To ensure that our header is included only once in the compilation process, we use the mechanism called header guards. It ensures that our header content is included only once in the compilation process. We surround the code in our header file with the following macros:

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

// header file source code
// goes here

#endif
```

3. Namespaces

A namespace is a scope with a name. To declare a namespace, we write:

```
namespace MyNameSpace
{

}
```

To declare objects in a namespace, we use:

```
namespace MyNameSpace
{
    int x;
    double d;
}
```

To refer to these objects outside the namespace, we use their fully qualified names.
This means we use the *namespace_name::our_object* notation.

```
namespace MyNameSpace
{
int x;
double d;
}
int main()
{
MyNameSpace::x = 123;
MyNameSpace::d = 456.789;
}
```

To introduce an entire namespace into the current scope, we can use the using
-directive:

```
namespace MyNameSpace
{
int x;
double d;
}
using namespace MyNameSpace;
int main()
{
x = 123;
d = 456.789;
}
```

`namespace MyNameSpace`
`{`
`int x;`
`double d;`
`}`
`namespace MyNameSpace`
`{`
`char c;`
`bool b;`
`}`
`int main()`
`{`
`MyNameSpace::x = 123;`
`MyNameSpace::d = 456.789;`
`MyNameSpace::c = 'a';`
`MyNameSpace::b = true;`
`}`
	We now have x, d, c, and b inside our MyNameSpace namespace. We are *extending* the MyNameSpace, not redefining it.
	A namespace can be spread across multiple files, both headers and source files. We will often see production code wrapped into namespaces. It is an excellent mechanism to group the code into namespaces logically.

`#include <iostream>`
`namespace MyNameSpace`
`{`
`int x;`
`}`
`Chapter 31 Organizing COde169`
`namespace MySecondNameSpace`
`{`
`int x;`
`}`
`int main()`
`{`
`MyNameSpace::x = 123;`
`MySecondNameSpace::x = 456;`
`std::cout << "1st x: " << MyNameSpace::x << ", 2nd x: " <<`
`MySecondNameSpace::x;`
`}`
	Two namespaces with different names can hold an object with the same name. Since every namespace is a different scope, they now declare two different unrelated objects with the same name

Chapter 33: Conversions

1. Implicit Conversions

Some values can be implicitly converted into each other. This is true for all the built-in types. We can convert char to int, int to double, etc

-

```
int main()
{
    char mychar = 64;
    int myint = 123;
    double mydouble = 456.789;
    bool myboolean = true;

    myint = mychar;
    mydouble = myint;
    mychar = myboolean;
}
```

- We can convert double to int but the decimal part is gone, this is called narrowing conversions
- When smaller integer types such as char or short are used in arithmetic operations, they get promoted/converted to integers.
- This is referred to as integral promotion

```
int main()
{
    char c1 = 10;
    char c2 = 20;

    auto result = c1 + c2; // result is of type int
}
```

Any built-in type can be converted to boolean. For objects of those types, any value other than 0, gets converted to a boolean value of true, and values equal to 0, implicitly convert to a value of false

```
int main()
{
    char mychar = 64;
    int myint = 0;
    double mydouble = 3.14;
    bool myboolean = true;

    myboolean = mychar;    // true
    myboolean = myint;     // false
    myboolean = mydouble;  // true
}
```

Conversely, a boolean type can be converted to int. The value of true converts to integer value 1 and the value of false converts to integer value of 0

- A pointer of any type can be converted to void* type:

```
int main()
{
    int x = 123;
    int* pint = &x;
    void* pvoid = pint;
}
```

- Arrays are implicitly convertible to pointers :

```
#include <iostream>

int main()
{
    int arr[5] = { 1, 2, 3, 4, 5 };
    int* p = arr; // pointer to the first array element
    std::cout << *p;
}
```

In this case, we have an implicit conversion of type *int[]* to type *int**

```
#include <iostream>

void myfunction(int arg[])
{
    std::cout << *arg;
}

int main()
{
    int arr[5] = { 1, 2, 3, 4, 5 };
    myfunction(arr);
}
```

Here, the *arr* argument gets converted to a pointer to the first element in an array

Since arg is now a pointer, printing it outputs a pointer value similar to the *adress*

2. Explicit Conversions

Syntax:

```
int main()
{
    auto myinteger = static_cast<int>(123.456);
}
```

Prefer this verbose function to implicit conversions, as the static_cast is the idiomatic way of converting between convertible types. This function performs a
compile-time conversion

```cpp
#include <iostream>
class MyBaseClass {
public:
virtual ~MyBaseClass() {}
};
class MyDerivedClass : public MyBaseClass {};
int main()
{
MyBaseClass* base = new MyDerivedClass;
MyDerivedClass* derived = new MyDerivedClass;
// base to derived
if (dynamic_cast<MyDerivedClass*>(base))
{
std::cout << "OK.\n";
}
else
{
std::cout << "Not convertible.\n";
}

// derived to base
if (dynamic_cast<MyBaseClass*>(derived))
{
std::cout << "OK.\n";
}
else
{
std::cout << "Not convertible.\n";
}
delete base;
delete derived;
}
```

```
OK.
```

The following explicit conversion functions should be used **rarely** and carefully.
They are dynamic_cast and reintepret_cast. The dynamic_cast function converts pointers of base class to pointers to derived class and vice versa up the inheritance chain