Chapter 27

The static specifier

- The object will have a **static storage duration**
- Memory space for static objects is **allocated when the program starts** and **deallocated when the program ends**.
- **One instance** of a static object exists in the program
- Space for a local variable is **allocated** the first time the program control encounters its definition and **deallocated** when the program exits, when it is marked as **static**.

```cpp
#include <iostream>
void myfunction()
{
    static int x = 0; // defined only the first time, skipped every other time
    x++; // increased x by 1
    std::cout << x << '\n';
}

int main()
{
    myfunction(); // x == 1
    myfunction(); // x == 2
    myfunction(); // x == 3
}
```

```
1
2
3
[Finished in 1.5s]
```

**** Do not have to store the value inside some global variable**

Static Class

- Static class members are **not part of the object**
- **Declare** a static data member **inside the class** and **define** it **outside the class only once**

```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[Finished in 1.7s]
```

Define a static member function

- prepend the **function declaration** with the **static** keyword
- function definition **outside the class** does not use the **static** keyword

```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.[Finished in 1.3s]
```

Chapter 28

Templates

- mechanisms to support the so-called **generic programming** ~ define a function or a class regardless of what types it accepts.
- when instantiate functions and classes, use **concrete type**
- Use when define a class or a function that **accepts almost any type**

```cpp
#include <iostream>
/* define a template, with T as placeholder
for a specific type */
template <typename T>
void myfunction(T param)
{
    std::cout << "The value of a parameter is: " << param << '\n';
}

int main()
{
    myfunction<int>(123);
    myfunction<double>(123.456);
    myfunction<char>('A');
}
```

```
The value of a parameter is: 123
The value of a parameter is: 123.456
The value of a parameter is: A
[Finished in 1.4s]
```

Multiple parameter

- simply list the template parameters and separate them using **a comma**

```cpp
#include <iostream>
// define a template that has 2 template parameters
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
[Finished in 1.6s]
```

Define a class template

```cpp
#include <iostream>
template <typename T>
class MyClass
//  defined a simple class template that accepts types T
{
private:
    T x;
public:
    MyClass(T xx)
    :x{ xx }
    { }
    T getvalue()
    {
        return x;
    }
};

int main()
{
    MyClass<int> o{ 123 };
    //  instantiate class with concrete type int
    std::cout << "The value of x is: " << o.getvalue() << '\n';
    MyClass<double> o2{ 456.789 };
    //  instantiate class with concrete type double
    std::cout << "The value of x is: " << o2.getvalue() << '\n';
}
```

```
The value of x is: 123
The value of x is: 456.789
[Finished in 1.2s]
```

- Instead of having to write the same code for two or more different types, **use a template**

Define a class template member functions outside the class

- Make them templates themselves by **prepending the member function definition** with the appropriate **template declaration**
- a class name **MUST** be called with **a template argument**

```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 };
}
```

```
Constructor invoked. The value of x is: 123
Constructor invoked. The value of x is: 456.789
[Finished in 1.4s]
```

Template specialization

- Make template to **behave differently** for a specific type (sometimes a different code)

```
#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)
// specialize the function myfunction() with type int
{
    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
}
```

```
The value of an argument is: A
The value of an argument is: 345.678
This is a specialization int. The value is: 123
[Finished in 1.3s]
```

Chapter 29

Enumerations

- a type whose values are **user-defined named constants**
- two kinds of enums: the **unscoped enums** and **scoped enums.**

Unscoped enums

- have their enumerators **leak into an outside scope**, where the enum type itself defined
- Old enums are **best avoided**

```
#include <iostream>
using namespace std;
enum MyEnum
{
    myfirstvalue = 10,
    mysecondvalue = 15,
    mythirdvalue = 30,
};

int main()
{
    MyEnum myenum = myfirstvalue;
    cout << myenum << '\n';
    myenum = mysecondvalue; // we can change the value of our enum object
    cout << myenum;
}
```

```
10
15[Finished in 1.5s]
```

Scoped enums

- **do not leak** their enumerators into an outer scope
- are **not implicitly convertible** to other types

```
enum class MyEnum
// define a scoped enum
{
        myfirstvalue,
        mysecondvalue,
        mythirdvalue
};

int main()
{
        MyEnum myenum = MyEnum::myfirstvalue;
        // declare a variable of type enum class
}
```

- To access an enumerator value

```
MyEnum::myfirstvalue
```

- the enumerator names are defined **only within the enum internal scope** and **implicitly convert to underlying types**
- specify the underlying type for scoped enum

```
enum class MyCharEnum : [type]
```

Conclusion

- prefer **enum class enumerations (scoped enums)** to old plain **unscoped enums**
- Use enumerations when our object is to have one value out of **a set of predefined named values**

Chapter 31

Organizing code

- can split our C++ code **into multiple files.**
- two kinds of files into which we can store our C++ source: **header files (headers)** and **source files.**

31.1 Header and Source Files

Header

- Are **source code files** where we usually put **various declarations**
- Have the **.h (or .hpp) extension.**

Source files

- are files where we can **store our definitions and the main program.**
- have the **.cpp (or .cc) extension.**

To include a standard library header

```
#include <headername>
```

To include user-defined header files

```
#include “headername”
```

- Sometimes we need to include **both standard-library headers and user-defined headers**

```
#include <iostream>
#include "myheader.h”
```

Translation unit

- **The compiler stitches** the code from the header file and the source file together
- The compiler then uses this file to **create an object file**
- **A linker** then links object files together to **create a program.**

Should

- put **the declarations and constants** into header files
- put **definitions and executable code** in source files

31.2 Header Guards

- Multiple source files might include the same header file
- To ensure that header content is **included only once** in **the compilation process**

```
#ifndef MY_HEADER_H
#define MY_HEADER_H
/* header file source code
goes here */
#endif
```

31.3 Namespaces

- logically **group parts of the code**
- A namespace is **a scope with a name**

```
namespace MyNameSpace
{
    int x;
    double d;
    // declare objects in a namespace
}

int main()
{
    MyNameSpace::x = 123;
    MyNameSpace::d = 456.789;
    // define the objects outside the namespace
}
```

To introduce an entire namespace into the current scope

```
namespace MyNameSpace
{
    int x;
    double d;
    // declare objects in a namespace
}

namespace MyNameSpace
{
    char c;
    bool b;
}

using namespace MyNameSpace
// introduce an entire namespace into the current scope

int main()
{
    x = 123;
    d = 456.789;
    c = 'a';
    b = true;
    // define the objects outside the namespace
}
```

- If we have **several separate namespaces** with **the same name** in our code, this means we are **extending that namespace**
- A namespace can be spread across multiple files, **both headers and source files** = an excellent mechanism **to group the code into namespaces logically**
- 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**

```
#include <iostream>
namespace MyNameSpace
{
    int x;
}

namespace MySecondNameSpace
{
    int x;
}

int main()
{
    MyNameSpace::x = 123;
    MySecondNameSpace::x = 456;
    std::cout << "1st x: " << MyNameSpace::x << ", 2nd x: " << MySecondNameSpace::x;
}
```

```
1st x: 123, 2nd x: 456[Finished in 1.2s]
```

Chapter 33
Conversions

- Some values can be **implicitly converted into each other**, true for all the built-in types

Narrowing conversions

- Implicitly convert double to int => the lost of information (**narrowing conversions**)

```
int main()
{
    int myint = 123;
    double mydouble = 456.789;
    myint = mydouble; // the decimal part is lost
}
```

Integral promotion

- When **smaller integer types** such as **char or short** are used in **arithmetic operations**, they get **promoted/converted to integers**

```
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**
- 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
}
```

- 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**

Pointers

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

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

- can **convert any data pointer to a void pointer** but we can **not dereference the void pointer.**
- To be able to access the object pointed to by a void pointer, we need to **cast the void pointer to some other pointer type first** = use the **explicit cast function static_cast**

```
#include <iostream>
using namespace std;

int main()
{
    int x = 123;
    int* pint = &x;
    void* pvoid = pint; // convert from int pointer
    int* pint2 = static_cast<int*>(pvoid); // cast a void pointer to int pointer
    std::cout << *pint2; // dereference a pointer
}
```

```
123[Finished in 1.1s]
```

Arrays

- are **implicitly convertible to pointers.**
- assign an array name to the pointer, the pointer **points at the first element in an array.**

```
#include <iostream>
int main()
{
    int arr[5] = { 1, 2, 3, 4, 5 };
    int* p = arr;
    /* points to the first array element,
    implicit conversion of type int[] to type int*. */
    std::cout << *p;
}
```

- When used **as function arguments**, the array gets **converted to a pointer.**

=> The array **loses its dimension**, **decays to a pointer**.

```cpp
#include <iostream>
void myfunction(int arg[])
{
    std::cout << arg;
}

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

```
0x61ff0c[Finished in 1.3s]
```

- the arr argument gets **converted to a pointer to the first element in an array.**
- arg is now **a pointer**, printing it outputs a pointer **value similar to the 012FF6D8**

```cpp
#include <iostream>
void myfunction(int arg[])
{
    std::cout << *arg; // output the value the pointer points to
}

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

```
1[Finished in 1.3s]
```

=> important to adopt the following: **prefer std:vector and std::array containers** to raw arrays and pointers

33.2 Explicit Conversions

- can **explicitly convert the value of one type to another**
- **static_cast function** converts between **implicitly convertible types**

```
static_cast<type_to_convert_to>(value_to_convert_from)
```

- the static_cast is **the idiomatic way** of **converting between convertible types**
- performs **a compile-time conversion**

Dynamic_cast

- function c**onverts pointers of base class to pointers** to derived class and vice versa up the inheritance chain

```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.
OK.
[Finished in 1.0s]
```

Success	Not success
result = a pointer to **a base or derived class**	result = a pointer of **value nullptr**

- To use this function, our class **MUST be polymorphic (base class should have at least one virtual function)**

```cpp
#include <iostream>
class MyBaseClass {
public:
virtual ~MyBaseClass()
{}
};

class MyDerivedClass : public MyBaseClass {};
class MyUnrelatedClass {};

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

```
Not convertible.
Not convertible.
[Finished in 1.3s]
```

- fail as the dynamic_cast can **only convert between related classes inside the inheritance chain**
- **hardly ever** have to use dynamic_cast

Reintrepret_cast

- The most dangerous cast
- is **best avoided** as it does **not offer guarantees** of any kind

=> the **static_cast function** is probably the **only cast** we will be using **most of the time.**