# Declarations and Definitions

Declarations also serve as definitions(initiate with a value), except when the declaration:

1. Is a function prototype (a function declaration with no function body).
2. Contains the **extern** specifier but no initializer (objects and variables) or function body (functions). This signifies that the definition is not necessarily in the current translation unit and gives the name external linkage.
3. Is of a static data member inside a class declaration.

Because static class data members are discrete variables shared by all objects of the class, they must be defined and initialized outside the class declaration. (For more information about classes and class members, see [Classes](https://msdn.microsoft.com/en-us/library/4a1hcx0y.aspx).)

1. Is a class name declaration with no following definition, such as class T;.
2. Is a **typedef** statement.

When modifying a data declaration, the **const** keyword specifies that the object or variable is not modifiable.

const int maxarray = 255;

char store\_char[maxarray];

A **pointer** to a variable declared as **const** can be assigned only to a pointer that is also declared as **const**.

You can use **pointers to constant data** as function parameters to prevent the function from modifying a parameter passed through a pointer.

For **objects** that are declared as **const**, you can only call [constant member functions](https://msdn.microsoft.com/en-us/library/6ke686zh.aspx). This ensures that the constant object is never modified.

Declaring a **member function** with the **const** keyword specifies that the function is a "read-only" function that does not modify the object for which it is called. The **const** keyword is required in both the declaration and the definition.

**extern** const int i = 2;

Use this variable in another module like:

**extern** const int i;

[**Default initialization of constant variables**](javascript:void(0))

Constant variables must be declared together with an initializer. If they are scalar types they cause a compiler error, and if they are class types that have a default constructor they cause a warning:

class MyClass{};

int main() {

//const int i2; // compiler error C2734: const object must be initialized if not extern

//const char c2; // same error

const MyClass mc1; // compiler error C4269: 'const automatic data initialized with compiler generated default constructor produces unreliable results

}

Static variables that are declared with no initializer are initialized to 0 (implicitly converted to the type).

class MyClass {

private:

int m\_int;

char m\_char;

};

int main() {

static int int1; // 0

static char char1; // '\0'

static bool bool1; // false

static MyClass mc1; // {0, '\0'}

}

using identifier = type;

// the same as typedef

using func = void(\*)(int);

// C++03 equivalent:

// typedef void (\*func)(int);

// func can be assigned to a function pointer value

void actual\_function(int arg) { /\* some code \*/ }

func fptr = &actual\_function;

using declaration:

The **using** declaration introduces a name into the declarative region in which the **using** declaration appears.

class D : B {

public:

using B::f; // f later refers to B::f

using B::g;

Variables defined within a block have automatic storage unless otherwise specified using the **extern**, **static**, or **thread\_local** specifiers. Automatic objects and variables have no linkage; they are not visible to code outside the block.

**Static duration** means that the object or variable is allocated when the program starts and is deallocated when the program ends. **External linkage** means that the name of the variable is visible from outside the file in which the variable is declared. Conversely, **internal linkage** means that the name is not visible outside the file in which the variable is declared.

By default, an object or variable that is defined in the global namespace has static duration and external linkage. The **static** keyword can be used in the following situations.

1. At global level: static makes a variable have internal linkage.
2. At function level: static makes a variable maintain its state between function calls.
3. At class level: static data member is shared by all the instances. static funciton member only has access to static member.

Objects and variables declared as **extern** declare an object that is defined in another translation unit or in an enclosing scope as having **external linkage**.

In C++11, the **register** keyword is deprecated. It specifies that the variable is to be stored in a machine register, if possible.

**storage class**

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C++ Program. The **auto** storage class is the default storage class for all local variables. The **register** storage class is used to define local variables that should be stored in a register instead of RAM. The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope.

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files.

**Scope**

**Local scope Function scope** Namespace Scope **Class scope**

**Hide Names**

// file\_scopes.cpp

// compile with: /EHsc

#include <iostream>

int i = 7; // i has file scope, outside all blocks

using namespace std;

int main( int argc, char \*argv[] ) {

int i = 5; // i has block scope, hides i at file scope, ::i refers to i in global scope;

cout << "Block-scoped i has the value: " << i << "\n";

cout << "File-scoped i has the value: " << ::i << "\n";

}

# Lambda Expression

#include <algorithm>

#include <cmath>

void abssort(float\* x, unsigned n) {

std::sort(x, x + n,

// Lambda expression begins

[](float a, float b) {

return (std::abs(a) < std::abs(b));

} // end of lambda expression

);

}

# vector<int>::iterator

iterator is used to specify a range in c++. iterator is more helpful in c++ than index(int) since many build-in functions are based on iterators.

vector<int> arr;

arr.end() refers to the past-the-end element, shall not be dereferenced.

The ranges used by functions in standard library do not include the element pointed by the enclosing iterator.

std::find(arr.begin(), arr.end(), val); // find val in [begin,end）, end is exclued.

std::count(begin, end, val)

std::copy(begin, end, another\_begin)

std::remove(begin, end, val)

std::sort(begin, end)

std::reverse(begin, end)

Compare to C#,

Array.Find(arr, predict) Array.FindIndex(arr, predict)

Array.Sort()

Array.Reverse()

Array.Copy(source, destination, len)

source.CopyTo(destination, startIndex)

std::vector<int> v { 34,23 };

// or

// std::vector<int> v = { 34,23 };

is ok in the new c++ compiler

# Overloading operator

c#: public T this[K key]

{ get {} set{}}

c++: public T& operator[](K key) const

{ return }

# vector assignment

vector<int> v1, v2;

// filling v1

v2 = v1;

std::copy(v1.begin(), v1.end(), v2.begin());

//copy won’t work well if v1 and v2 do not have the same size

// assignment is ok. the operation on v1 does not influence v2.

vector<int> v2(v1);

# File Operation

## ifstream

// associate with a file

ifstream inFile(const char\* fileName);

ifstream inFile;

inFile.open(const char\* fileName); // both are right

// read from a file

while(inFile.good())

{

char c = inFile.get(); // or inFile >> c;

string s = inFile.getLine();

}

// close a flie

inFile.close()

## ofstream

ofstream outFile(const char\* fileName);

char c;

string s;

do

{

c = cin.get();

s = cin.getline();

outFile.put(c);

outFile.put(s, s.size());

}while(c != ‘.’);

cout 是ostream 类的一个实例

ostream类定义了<<方法，还提供.put() .write()方法

cout << i;

cout.put(const char c);

cout.write(const char\* s, int size);

ofstream inherits from ostream

if you ceate an instance of ofstream and associate with a file like:

ofstream outFile(const char\* fileName);

outFile has method including <<, .put(), .write()

cin is an instance of istream

How cin >> x behaves depends on the type of x.

Generally, it reads from non-empty character until the first unsatisfying character.

int x;

cin >> x;

if the input is 123Z. Then x will be 123;

cin.get() will not jump over empty character(like space, ‘\n’, ‘\t’)

cin.getline() reads the entire line.