Storage class specifiers

The storage class specifiers are a part of the *decl-specifier-seq* of a name's declaration syntax. Together with the scope of the name, they control two independent properties of the name: its *storage duration* and its *linkage*.

- **auto** or (until C++11)no specifier - *automatic* storage duration.

- **register** - *automatic* storage duration. Also hints to the compiler to place the object in the processor's register. (deprecated) (until C++17)

- **static** - *static* or *thread* storage duration and *internal* linkage (or *external* linkage for static class members not in an anonymous namespace).
- **extern** - *static* or *thread* storage duration and *external* linkage.

- **thread_local** - *thread* storage duration. (since C++11)

- **mutable** - does not affect storage duration or linkage. See const/volatile for the explanation.

Only one storage class specifier may appear in a declaration except that thread_local may be combined with static or with extern (since C++11).

Explanation

1) The auto specifier was only allowed for objects declared at block scope or in function parameter lists. It indicated automatic storage duration, which is the default for these kinds of declarations. The meaning of this keyword was changed in C++11. (until C++11)

2) The register specifier is only allowed for objects declared at block scope and in function parameter lists. It indicates automatic storage duration, which is the default for these kinds of declarations. Additionally, the presence of this keyword may be used as a hint for the optimizer to store the value of this variable in a CPU register. This keyword was deprecated in C++11. (until C++17)

3) The static specifier is only allowed in the declarations of objects (except in function parameter lists), declarations of functions (except at block scope), and declarations of anonymous unions. When used in a declaration of a class member, it declares a static member. When used in a declaration of an object, it specifies static storage duration (except if accompanied by thread_local). When used in a declaration at namespace scope, it specifies internal linkage.

4) The extern specifier is only allowed in the declarations of variables and functions (except class members or function parameters). It specifies external linkage, and does not technically affect storage duration, but it cannot be used in a definition of an automatic storage duration object, so all extern objects have static or thread durations. In addition, a variable declaration that uses extern and has no initializer is not a definition.

5) The thread_local keyword is only allowed for objects declared at namespace scope, objects declared at block scope, and static data members. It indicates that the object has thread storage duration. It can be combined with static or extern to specify internal or external linkage (except for static data members which always have external linkage), respectively, but that additional static doesn't affect the storage duration. (since C++11)

Storage duration

All objects in a program have one of the following storage durations:

- ***automatic*** storage duration. The storage for the object is allocated at the beginning of the enclosing code block and deallocated at the end. All local objects have this storage duration, except those declared static, extern or thread_local.

- ***static*** storage duration. The storage for the object is allocated when the program begins and deallocated when the program ends. Only one instance of the object exists. All objects declared at namespace scope (including global namespace) have this storage duration, plus those declared with static or extern. See Non-local variables and Static local variables for details on initialization of objects with this storage duration.

- ***thread*** storage duration. The storage for the object is allocated when the thread begins and deallocated when the thread ends. Each thread has its own instance of the object. Only objects declared thread_local have this storage duration. thread_local can appear together with static or extern to adjust linkage. See Non-local variables and Static local variables for details on initialization of objects with this storage duration. (since C++11)

- ***dynamic*** storage duration. The storage for the object is allocated and deallocated upon request by using dynamic memory allocation functions. See new-expression for details on initialization of objects with this storage duration.

Linkage

A name that denotes object, reference, function, type, template, namespace, or value, may have *linkage*. If a name has linkage, it refers to the same entity as the same name introduced by a declaration in another scope. If a variable, function, or another entity with the same name is declared in several scopes, but does not have sufficient linkage, then several instances of the entity are generated.

The following linkages are recognized:

no linkage

The name can be referred to only from the scope it is in.

Any of the following names declared at block scope have no linkage:

- variables that aren't explicitly declared extern (regardless of the static modifier);

- local classes and their member functions;
- other names declared at block scope such as typedefs, enumerations, and enumerators.

Names not specified with external, module, (since C++20) or internal linkage also have no linkage, regardless of which scope they are declared in.

internal linkage

The name can be referred to from all scopes in the current translation unit.

Any of the following names declared at namespace scope have internal linkage:

- variables, variable templates (since C++14), functions, or function templates declared static;
- non-volatile non-template (since C++14) non-inline (since C++17) non-exported (since C++20) const-qualified variables (including constexpr) that aren't declared extern and aren't previously declared to have external linkage;
- data members of anonymous unions.

In addition, all names declared in unnamed namespace or a namespace within an unnamed namespace, even ones explicitly declared extern, have internal linkage. (since C++11)

external linkage

The name can be referred to from the scopes in the other translation units. Variables and functions with external linkage also have language linkage, which makes it possible to link translation units written in different programming languages.

Any of the following names declared at namespace scope have external linkage, unless they are declared in an unnamed namespace or their declarations are attached to a named module and are not exported (since C++20):

- variables and functions not listed above (that is, functions not declared static, non-const variables not declared static, and any variables declared extern);
- enumerations;
- names of classes, their member functions, static data members (const or not), nested classes and enumerations, and functions first introduced with friend declarations inside class bodies;
- names of all templates not listed above (that is, not function templates declared static).

Any of the following names first declared at block scope have external linkage:

- names of variables declared extern;
- names of functions.

module linkage

The name can be referred to only from the scopes in the same module unit or in the other translation units of the same named module. (since C++20)

Names declared at namespace scope have module linkage if their declarations are attached to a named module and are not exported, and don't have internal linkage.

This section is incomplete
Reason: add the description of the behavior when an entity is declared with different linkages in the same translation unit (6.6 paragraph 6), note the difference between C++20 (ill-formed) and the current draft (well-formed)

Static local variables

Variables declared at block scope with the specifier static or thread_local (since C++11) have static or thread (since C++11) storage duration but are initialized the first time control passes through their declaration (unless their initialization is zero- or constant-initialization, which can be performed before the block is first entered). On all further calls, the declaration is skipped.

If the initialization throws an exception, the variable is not considered to be initialized, and initialization will be attempted again the next time control passes through the declaration.

If the initialization recursively enters the block in which the variable is being initialized, the behavior is undefined.

If multiple threads attempt to initialize the same static local variable concurrently, the initialization occurs exactly once (similar behavior can be obtained for arbitrary functions with std::call_once). (since C++11)

Note: usual implementations of this feature use variants of the double-checked locking pattern, which reduces runtime overhead for already-initialized local statics to a single non-atomic boolean comparison.

The destructor for a block-scope static variable is called at program exit, but only if the initialization took place successfully.

Function-local static objects in all definitions of the same inline function (which may be implicitly inline) all refer to the same object defined in one translation unit, as long as the function has external linkage.

Translation-unit-local entities

The concept of translation-unit-local entities is standardized in C++20, see this page for more details.

An entity is *translation-unit-local* (or *TU-local* for short) if

- it has a name with internal linkage, or
- it does not have a name with linkage and is introduced within the definition of a TU-local entity, or
- it is a template or template specialization whose template argument or template declaration uses a TU-local entity.

Bad things (usually violation of ODR) can happen if the type of a non-TU-local entity depends on a TU-local entity, or if a declaration of, or a deduction guide for, (since C++17) a non-TU-local entity names a TU-local entity outside its

- function-body for a non-inline function or function template

- initializer for a variable or variable template
- friend declarations in a class definition
- use of value of a variable, if the variable is usable in constant expressions

Such uses are disallowed in a module interface unit (outside its private-module-fragment, if any) or a module partition, and are deprecated in any other context. A declaration that appears in one translation unit cannot name a TU-local entity declared in another translation unit that is not a header unit. A declaration instantiated for a template appears at the point of instantiation of the specialization.	(since C++20)

Notes

Names at the top-level namespace scope (file scope in C) that are const and not extern have external linkage in C, but internal linkage in C++.

Since C++11, auto is no longer a storage class specifier; it is used to indicate type deduction.

In C, the address of a register variable cannot be taken, but in C++, a variable declared register is semantically indistinguishable from a variable declared without any storage class specifiers.	(until C++17)
In C++, unlike C, variables cannot be declared register.	(since C++17)

Names of thread_local variables with internal or external linkage referred from different scopes may refer to the same or to different instances depending on whether the code is executing in the same or in different threads.

The extern keyword can also be used to specify language linkage and explicit template instantiation declarations, but it's not a storage class specifier in those cases (except when a declaration is directly contained in a language linkage specification, in which case the declaration is treated as if it contains the extern specifier).

The keyword mutable is a storage class specifier in the C++ language grammar, although it doesn't affect storage duration or linkage.

This section is incomplete
Reason: the rules about re-declaring names in the same TU

Storage class specifiers, except for thread_local, are not allowed on explicit specializations and explicit instantiations:

```
template<class T>
struct S
{
    thread_local static int tlm;
};

template<>
thread_local int S<float>::tlm = 0; // "static" does not appear here
```

Keywords

auto, register, static, extern, thread_local, mutable

Example

Run this code

```
#include <iostream>
#include <string>
#include <thread>
#include <mutex>

thread_local unsigned int rage = 1;
std::mutex cout_mutex;

void increase_rage(const std::string& thread_name)
{
    ++rage; // modifying outside a lock is okay; this is a thread-local variable
    std::lock_guard<std::mutex> lock(cout_mutex);
    std::cout << "Rage counter for " << thread_name << ": " << rage << '\n';
}

int main()
{
    std::thread a(increase_rage, "a"), b(increase_rage, "b");

    {
        std::lock_guard<std::mutex> lock(cout_mutex);
        std::cout << "Rage counter for main: " << rage << '\n';
    }

    a.join();
    b.join();
}
```

Possible output:

```
Rage counter for a: 2
Rage counter for main: 1
Rage counter for b: 2
```

Defect reports

The following behavior-changing defect reports were applied retroactively to previously published C++ standards.

DR	Applied to	Behavior as published	Correct behavior
CWG 216 (https://cplusplus.github.io/CWG/issues/216.html)	C++98	unnamed class and enumeration in class scope have different linkage from those in namespace scope	they all have external linkage in these scopes
CWG 389 (https://cplusplus.github.io/CWG/issues/389.html)	C++98	a name with no linkage should not be used to declare an entity with linkage	a type without linkage shall not be used as the type of a variable or function with linkage, unless the variable or function has C language linkage
CWG 426 (https://cplusplus.github.io/CWG/issues/426.html)	C++98	an entity could be declared with both internal and external linkage in the same translation unit	the program is ill-formed in this case
CWG 527 (https://cplusplus.github.io/CWG/issues/527.html)	C++98	the type restriction introduced by the resolution of CWG 389 was also applied to variables and functions that cannot be named outside their own translation units	the restriction is lifted for these variables and functions (i.e. with no linkage or internal linkage, or declared within unnamed namesapces)
CWG 2387 (https://cplusplus.github.io/CWG/issues/2387.html)	C++14	unclear whether const-qualified variable template have internal linkage by default	const qualifier does not affect the linkage of variable templates or their instances

See also

C documentation for **storage duration**

Retrieved from "https://en.cppreference.com/mwiki/index.php?title=cpp/language/storage_duration&oldid=138611"