[1 基本概念 2](#_Toc498373002)

[1.1 注释 3](#_Toc498373003)

[**1.1.1 语法** 3](#_Toc498373004)

[**1.1.2 C风格** 3](#_Toc498373005)

[**1.1.3 C++风格** 3](#_Toc498373006)

[**1.1.4 注意** 3](#_Toc498373007)

[1.2 ASCII表 4](#_Toc498373008)

[1.3 名称和标识符 4](#_Toc498373009)

[**1.3.1 标识符** 4](#_Toc498373010)

[**1.3.2 Names** 5](#_Toc498373011)

[1.4 Types - Fundamental types 6](#_Toc498373012)

[**1.4.1 Type classification** 6](#_Toc498373013)

[**1.4.2 Type naming** 7](#_Toc498373014)

[**1.4.3 Elaborated type specifier** 9](#_Toc498373015)

[**1.4.4 Static type** 9](#_Toc498373016)

[**1.4.5 Dynamic type** 9](#_Toc498373017)

[**1.4.6 Incomplete type** 9](#_Toc498373018)

[1.5 Object - Scope - Lifetime 14](#_Toc498373019)

[1.5.1 **Object** 14](#_Toc498373020)

[1.6 Definitions and ODR 18](#_Toc498373021)

[1.9 存储模型和数据竞争 18](#_Toc498373022)

[**1.9.1 字节** 19](#_Toc498373023)

[**1.9.2 存储单元** 19](#_Toc498373024)

[**1.9.3 线程和数据竞争** 19](#_Toc498373025)

[1.10 转译阶段 20](#_Toc498373026)

[**1.10.1 阶段1** 20](#_Toc498373027)

[**1.10.2 阶段2** 20](#_Toc498373028)

[**1.10.3 阶段3** 21](#_Toc498373029)

[**1.10.4 阶段4** 21](#_Toc498373030)

[**1.10.5 阶段5** 21](#_Toc498373031)

[**1.10.6 阶段6** 21](#_Toc498373032)

[**1.10.7 阶段7** 22](#_Toc498373033)

[**1.10.8 阶段8** 22](#_Toc498373034)

[**1.10.9 阶段9** 22](#_Toc498373035)

[**1.10.10 注意** 22](#_Toc498373036)

[**1.10.11 参考** 22](#_Toc498373037)

[1.11 The main() function 22](#_Toc498373038)

[5 声明 22](#_Toc498373039)

[5.7 存储周期和链接 22](#_Toc498373040)

[**5.7.1 存储类关键字** 22](#_Toc498373041)

[**5.7.2 解释** 23](#_Toc498373042)

[**5.7.3 存储期** 23](#_Toc498373043)

[**5.7.4 链接属性** 23](#_Toc498373044)

[**5.7.5 静态局部变量** 24](#_Toc498373045)

[**5.7.6 注意** 25](#_Toc498373046)

[**5.7.7 关键字** 25](#_Toc498373047)

[**5.7.8 举例说明** 26](#_Toc498373048)

[10 Templates 26](#_Toc498373049)

[10.9 Parameter pack 27](#_Toc498373050)

# **1 基本概念**

This section provides definitions for the specific terminology and the concepts used when describing the C++ programming language.

A C++ program is a sequence of text files (typically header and source files) that contain declarations. They undergo translation to become an executable program, which is executed when the OS calls its main function.

Certain words in a C++ program have special meaning, and these are known as keywords. Others can be used as identifiers. Comments are ignored during translation. Certain characters in the program have to be represented with escape sequences.

The entities of a C++ program are values, objects, references, functions, enumerators, types, class members, templates, template specializations, namespaces, parameter packs, and the “this” pointer. Preprocessor macros are not C++ entities.

Entities are introduced by declarations, which associate them with names and define their properties. The declarations that define all properties required to use an entity are definitions. A program must contain only one definition of any non-inline function or variable that is odr-used.

Definitions of functions include sequences of statements, some of which include expressions, which specify the computations to be performed by the program.

Names encountered in a program are associated with the declarations that introduced them using name lookup. Each name is only valid within a part of the program called its scope. Some names have linkage which makes them refer to the same entities when they appear in different scopes or translation units.

Each object, reference, function, expression in C++ is associated with a type, which may be fundamental, compound, or user-defined, complete or incomplete, etc.

Named objects and named references to objects are known as variables.

## **1.1 注释**

Comments serve as a sort of in-code documentation. When inserted into a program, they are effectively ignored by the compiler; they are solely intended to be used as notes by the humans that read source code. Although specific documentation is not part of the C++ standard, several utilities exist that parse comments with different documentation formats.

### **1.1.1 语法**

/\* 注释内容 \*/ (1)

// 注释内容 \n (2)

* Often known as "C-style" or "multi-line" comments.
* Often known as "C++-style" or "single-line" comments.

All comments are removed from the program at translation phase 3 by replacing each comment with a single whitespace character.

### **1.1.2 C风格**

C-style comments are usually used to comment large blocks of text, however, they can be used to comment single lines. To insert a C-style comment, simply surround text with /\* and \*/; this will cause the contents of the comment to be ignored by the compiler. Although it is not part of the C++ standard, /\*\* and \*/ are often used to indicate documentation blocks; this is legal because the second asterisk is simply treated as part of the comment. C-style comments cannot be nested.

### **1.1.3 C++风格**

C++-style comments are usually used to comment single lines, however, multiple C++-style comments can be placed together to form multi-line comments. C++-style comments tell the compiler to ignore all content between // and a new line.

### **1.1.4 注意**

Because comments are removed before the preprocessor stage, a macro cannot be used to form a comment and an unterminated C-style comment doesn't spill over from an #include'd file.

Besides commenting out, other mechanisms used for source code exclusion are

#if 0

std::cout << "this will not be executed or even compiled\n";

#endif

and

if(false) {

std::cout << "this will not be executed\n"

}

## **1.2 ASCII表**

The following chart contains all 128 ASCII decimal (dec), octal (oct), hexadecimal (hex) and character (ch) codes. The default is none.

## **1.3 名称和标识符**

### **1.3.1 标识符**

标识符由数字，下划线，小写字母和大写字母，和大部分的Unicode字符。合法的标识符不能以数字开头 ，而是以拉丁字母，下划线，或者Unicode编码的非数字字母开始。标识符是大小写敏感的，小写标识符和大写标识符是不同的。

Note: C++ grammar formally requires Unicode characters to be escaped with \u or \U, but due to translation phase 1, that is exactly how raw unicode characters from the source code are presented to the compiler. Also note that support of this feature may be limited, e.g. gcc.

**1.3.1.1 声明**

An identifier can be used to name objects, references, functions, enumerators, types, class members, namespaces, templates, template specializations, parameter packs, goto labels, and other entities, with the following exceptions:

* the identifiers that are keywords cannot be used for other purposes;
* the identifiers with a double underscore anywhere are reserved;
* the identifiers that begin with an underscore followed by an uppercase letter are reserved;
* the identifiers that begin with an underscore are reserved in the global namespace.

"Reserved" here means that the standard library headers #define or declare such identifiers for their internal needs, the compiler may predefine non-standard identifiers of that kind, and that name mangling algorithm may assume that some of these identifiers are not in use. If the programmer uses such identifiers, the behavior is undefined.

In addition, it's undefined behavior to #define or #undef names identical to keywords. If at least one standard library header is included, it's undefined behavior to #define or #undef identifiers identical to names declared in any standard library header.

*This section is incomplete*

*Reason: +other contents of 17.6.4.3 [reserved.names]*

**1.3.1.2 表达式**

An identifier that names a variable, a function, specialization of a concept, (since C++20) or an enumerator can be used as an expression. The expression consisting of just the identifier returns the entity named by the identifier. The value category of the expression is lvalue if the identifier names a function, a variable, or a data member, and prvalue otherwise (e.g. an enumerator is a prvalue expression, a specialization of a concept is a bool prvalue (since C++20)).

Within the body of a non-static member function, each identifier that names a non-static member is implicitly transformed to a class member access expression this->member.

**1.3.1.3 Unqualified identifiers**

Besides suitably declared identifiers, the following can be used in expressions in the same role:

* 函数表达式里的重载运算符，如***operator*+** 或 ***operator* new**；
* a user-defined conversion function name, such as ***operator bool***;
* a user-defined literal operator name, such as ***operator "" \_km***;
* a template name followed by its argument list, such as ***MyTemplate<int>***;
* the character ~ followed by a class name, such as ***~MyClass***;
* the character ~ followed by a decltype specifier, such as ***~decltype(str).***

Together with identifiers they are known as unqualified id-expressions.

**1.3.1.4 Qualified identifiers**

A qualified id-expression is an unqualified id-expression prepended by a scope resolution operator ::, and optionally, a sequence of enumeration, (since C++11)class or namespace names or decltype expressions (since C++11) separated by scope resolution operators. For example, the expression std::string::npos is an expression that names the static member npos in the class string in namespace std. The expression ::tolower names the function tolower in the global namespace. The expression ::std::cout names the global variable cout in namespace std, which is a top-level namespace. The expression boost::signals2::connection names the type connection declared in namespace signals2, which is declared in namespace boost.

The keyword template may appear in qualified identifiers as necessary to disambiguate dependent template names.

See qualified lookup for the details of the name lookup for qualified identifiers.

### **1.3.2 Names**

A name is the use of one of the following to refer to an entity or to a label:

* 标识符；
* 函数表达式里重载运算符的名称 (operator+, operator new);
* 用户自定义的类型转换函数 (operator bool);
* a user-defined literal operator name (operator "" \_km);
* a template name followed by its argument list (MyTemplate<int>).

Every name that denotes an entity is introduced into the program by a declaration. Every name that denotes a label is introduced into the program either by a goto statement or by a labeled statement. A name used in more than one translation unit may refer to the same or different entities, depending on linkage.

When the compiler encounters an unknown name in a program, it associates it with the declaration that introduced the name by means of name lookup, except for the dependent names in template declarations and definitions (for those names, the compiler determines whether they name a type, a template, or some other entity, which may require explicit disambiguation).

## **1.4 Types - Fundamental types**

Objects, references, functions including function template specializations, and expressions have a property called type, which both restricts the operations that are permitted for those entities and provides semantic meaning to the otherwise generic sequences of bits.

### **1.4.1 Type classification**

The C++ type system consists of the following types:

* fundamental types (see also std::is\_fundamental):
* the type void (see also std::is\_void);
* the type std::nullptr\_t (since C++11) (see also std::is\_null\_pointer);
* arithmetic types (see also std::is\_arithmetic):
* floating-point types (float, double, long double) (see also std::is\_floating\_point);
* integral types (see also std::is\_integral):
* the type bool;
* character types:
* narrow character types (char, signed char, unsigned char);
* wide character types (char16\_t, char32\_t, wchar\_t);
* signed integer types (short int, int, long int, long long int);
* unsigned integer types (unsigned short int, unsigned int, unsigned long int, unsigned long long int);
* compound types (see also std::is\_compound):
* reference types (see also std::is\_reference):
  + lvalue reference types (see also std::is\_lvalue\_reference):
    - lvalue reference to object types;
    - lvalue reference to function types;
* rvalue reference types (see also std::is\_rvalue\_reference):
* rvalue reference to object types;
* rvalue reference to function types;
* pointer types (see also std::is\_pointer):
* pointer to object types;
* pointer to function types;
* pointer to member types (see also std::is\_member\_pointer):
* pointer to data member types (see also std::is\_member\_object\_pointer);
* pointer to member function types (see also std::is\_member\_function\_pointer);
* array types (see also std::is\_array);
* function types (see also std::is\_function);
* enumeration types (see also std::is\_enum);
* class types:
* non-union types (see also std::is\_class);
* union types (see also std::is\_union).

For every type other than reference and function, the type system supports three additional cv-qualified versions of that type (const, volatile, and const volatile).

Types are grouped in various categories based on their properties:

* object type is a (possibly cv-qualified) type that is not a function type, not a reference type, and not void type (see also ***std::is\_object***);
* scalar types are (possibly cv-qualified) arithmetic, pointer, pointer to member, enumeration, and std::nullptr\_t types (see also std::is\_scalar);
* trivial types (see also std::is\_trivial), POD types (see also std::is\_pod), literal types (see also std::is\_literal\_type), and other categories listed in the the type traits library or as named type requirements.

### **1.4.2 Type naming**

A name can be declared to refer to a type by means of:

* class declaration;
* enum declaration;
* typedef declaration;
* type alias declaration.

Types that do not have names often need to be referred to in C++ programs; the syntax for that is known as type-id. The syntax of the type-id that names type T is exactly the syntax of a declaration of a variable or function of type T, with the identifier omitted, except that decl-specifier-seq of the declaration grammar is constrained to type-specifier-seq, and that new types may be defined only if the type-id appears on the right-hand side of a non-template type alias declaration.

int\* p; // 指向整数的指针声明

static\_cast<int\*>(p); // type-id是"int\*"

int a[3]; // 三个int型元素的数组

new int[3]; // type-id是"int[3]" (called new-type-id)

int (\*(\*x[2])())[3]; // 2个指向函数的指针元素的数组

// 返回指向3个int元素的数组

new (int (\*(\*[2])())[3]); // type-id是"int (\*(\*[2])())[3]"

void f(int); // 函数的声明，形参为int，返回值类型为void

std::function<void(int)> x = f; // 类型模板参数是type-id "void(int)"

std::function<auto(int) -> void> y = f; // same

std::vector<int> v; // declaration of a vector of int

sizeof(std::vector<int>); // type-id is "std::vector<int>"

struct { int x; } b; // creates a new type and declares an object b of that type

sizeof(struct{ int x; }); // error: cannot define new types in a sizeof expression

using t = struct { int x; }; // creates a new type and declares t as an alias of that type

sizeof(static int); // error: storage class specifiers not part of type-specifier-seq

std::function<inline void(int)> f; // error: neither are function specifiers

The declarator part of the declaration grammar with the name removed is referred to as abstract-declarator.

Type-id may be used in the following situations:

* to specify the target type in cast expressions;
* as arguments to sizeof, alignof, alignas, new, and typeid;
* on the right-hand side of a type alias declaration;
* as the trailing return type of a function declaration;
* as the default argument of a template type parameter;
* as the template argument for a template type parameter;
* in dynamic exception specification.

Type-id can be used with some modifications in the following situations:

* in the parameter list of a function (when the parameter name is omitted), type-id uses decl-specifier-seq instead of type-specifier-seq (in particular, some storage class specifiers are allowed);
* in the name of a user-defined conversion function, the abstract declarator cannot include function or array operators.

This section is incomplete

Reason: 8.2[dcl.ambig.res] if it can be compactly summarized

This section is incomplete

Reason: mention and link to decltype and auto

### **1.4.3 Elaborated type specifier**

Elaborated type specifiers may be used to refer to a previously-declared class name (class, struct, or union) or to a previously-declared enum name even if the name was hidden by a non-type declaration. They may also be used to declare new class names.

See elaborated type specifier for details.

### **1.4.4 Static type**

The type of an expression that results from the compile-time analysis of the program is known as the static type of the expression. The static type does not change while the program is executing.

### **1.4.5 Dynamic type**

If some glvalue expression refers to a polymorphic object, the type of its most derived object is known as the dynamic type.

// given

struct B { virtual ~B() {} }; // 多态类型

struct D: B {}; // 多态类型

D d; // 最底层的对象（most-derived object）

B\* ptr = &d;

// the static type of (\*ptr) is B

// the dynamic type of (\*ptr) is D

For prvalue expressions, the dynamic type is always the same as the static type.

### **1.4.6 Incomplete type**

The following types are incomplete types:

* the type void (possibly cv-qualified);
* class type that has been declared (e.g. by forward declaration) but not defined;
* array of unknown bound;
* array of elements of incomplete type;
* enumeration type from the point of declaration until its underlying type is determined.

Any of the following contexts requires class T to be complete:

* definition or function call to a function with return type T or argument type T;
* definition of an object of type T;
* declaration of a non-static class data member of type T;
* new-expression for an object of type T or an array whose element type is T;
* lvalue-to-rvalue conversion applied to a glvalue of type T;
* an implicit or explicit conversion to type T;
* a standard conversion, dynamic\_cast, or static\_cast to type T\* or T&, except when converting from the null pointer constant or from a pointer to void;
* class member access operator applied to an expression of type T;
* typeid, sizeof, or alignof operator applied to type T;
* arithmetic operator applied to a pointer to T;
* definition of a class with base class T;
* assignment to an lvalue of type T;
* a catch-clause for an exception of type T, T&, or T\*.

(In general, when the size and layout of T must be known.)

If any of these situations occur in a translation unit, the definition of the type must appear in the same translation unit. Otherwise, it is not required.

This section is incomplete

Reason: rules for completing the incomplete types from §3.9[basic.types]/6

1.4.7 Fundamental types

(See also type for type system overview and **the list of type-related utilities** that are provided by the C++ library)

1.4.7.1 void type

void - type with an empty set of values. It is an incomplete type that cannot be completed (consequently, objects of type void are disallowed). There are no arrays of void, nor references to void. However, pointers to void and functions returning type void (procedures in other languages) are permitted.

std::nullptr\_t

Defined in header <cstddef>

typedef decltype(nullptr) nullptr\_t; (since C++11)

std::nullptr\_t is the type of the null pointer literal, nullptr. It is a distinct type that is not itself a pointer type or a pointer to member type.

1.4.7.2 Boolean type

bool - type, capable of holding one of the two values: true or false. The value of sizeof(bool) is implementation defined and might differ from 1.

1.4.7.3 Character types

signed char - type for signed character representation.

unsigned char - type for unsigned character representation. Also used to inspect object representations (raw memory).

char - type for character representation which can be most efficiently processed on the target system (has the same representation and alignment as either signed char or unsigned char, but is always a distinct type). Multibyte characters strings use this type to represent code units. The character types are large enough to represent any UTF-8 code unit (since C++14). The signedness of char depends on the compiler and the target platform: the defaults for ARM and PowerPC are typically unsigned, the defaults for x86 and x64 are typically signed.

wchar\_t - type for wide character representation (see wide strings). Required to be large enough to represent any supported character code point (32 bits on systems that support Unicode. A notable exception is Windows, where wchar\_t is 16 bits and holds UTF-16 code units) It has the same size, signedness, and alignment as one of the integer types, but is a distinct type.

char16\_t - type for UTF-16 character representation, required to be large enough to represent any UTF-16 code unit (16 bits). It has the same size, signedness, and alignment as std::uint\_least16\_t, but is a distinct type. (since C++11)

char32\_t - type for UTF-32 character representation, required to be large enough to represent any UTF-32 code unit (32 bits). It has the same size, signedness, and alignment as std::uint\_least32\_t, but is a distinct type. (since C++11)

1.4.7.3 Integer types

int - basic integer type. The keyword int may be omitted if any of the modifiers listed below are used. If no length modifiers are present, it's guaranteed to have a width of at least 16 bits. However, on 32/64 bit systems it is almost exclusively guaranteed to have width of at least 32 bits (see below).

1.4.7.4 Modifiers

Modifies the integer type. Can be mixed in any order. Only one of each group can be present in type name.

1.4.7.5 Signedness

signed - target type will have signed representation (this is the default if omitted)

unsigned - target type will have unsigned representation

1.4.7.6 Size

short - target type will be optimized for space and will have width of at least 16 bits.

long - target type will have width of at least 32 bits.

long long - target type will have width of at least 64 bits.(since C++11)

Note: as with all type specifiers, any order is permitted: unsigned long long int and long int unsigned long name the same type.

Properties

The following table summarizes all available integer types and their properties:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Type specifier | Equivalent type | Width in bits by data model | | | | |
| C++ standard | LP32 | ILP32 | LLP64 | LP64 |
| short | short int | at least 16 | 16 | 16 | 16 | 16 |
| short int |
| signed short |
| signed short int |
| unsigned short | unsigned short int |
| unsigned short int |
| int | int | at least 16 | 16 | 32 | 32 | 32 |
| signed |
| signed int |
| unsigned | unsigned int |
| unsigned int |
| long | long int | at least 32 | 32 | 32 | 32 | 64 |
| long int |
| signed long |
| signed long int |
| unsigned long | unsigned long int |
| unsigned long int |
| long long | long long int  (C++11) | at least 64 | 64 | 64 | 64 | 64 |
| long long int |
| signed long long |
| signed long long int |
| unsigned long long | unsigned long long int(C++11) |
| unsigned long long int |

Besides the minimal bit counts, the C++ Standard guarantees that

1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)

Note: this allows the extreme case in which bytes are sized 64 bits, all types (including char) are 64 bits wide, and sizeof returns 1 for every type.

Note: integer arithmetic is defined differently for the signed and unsigned integer types. See arithmetic operators, in particular integer overflows.

1.4.7.7 Data models

The choices made by each implementation about the sizes of the fundamental types are collectively known as data model. Four data models found wide acceptance:

32 bit systems:

* LP32 or 2/4/4 (int is 16-bit, long and pointer are 32-bit)
* Win16 API
* ILP32 or 4/4/4 (int, long, and pointer are 32-bit);
* Win32 API
* Unix and Unix-like systems (Linux, Mac OS X)

64 bit systems:

* LLP64 or 4/4/8 (int and long are 32-bit, pointer is 64-bit)
* Win64 API
* LP64 or 4/8/8 (int is 32-bit, long and pointer are 64-bit)
* Unix and Unix-like systems (Linux, Mac OS X)

Other models are very rare. For example, ILP64 (8/8/8: int, long, and pointer are 64-bit) only appeared in some early 64-bit Unix systems (e.g. Unicos on Cray).

## 1.5 **Object - Scope - Lifetime**

### 1.5.1 **Object**

C++ programs create, destroy, refer to, access, and manipulate objects.

An object, in C++, is a region of storage that has

* size (can be determined with sizeof);
* alignment requirement (can be determined with alignof);
* storage duration (automatic, static, dynamic, thread-local);
* lifetime (bounded by storage duration or temporary);
* type;
* value (which may be indeterminate, e.g. for default-initialized non-class types);
* optionally, a name.

The following entities are not objects: value, reference, function, enumerator, type, non-static class member, bit-field, template, class or function template specialization, namespace, parameter pack, and this.

A variable is an object or a reference that is not a non-static data member, that is introduced by a declaration.

Objects are created by definitions, new-expressions, throw-expressions, when changing the active member of a union, and where temporary objects are required.

1.5.1.1 Object representation and value representation

For an object of type T, object representation is the sequence of sizeof(T) objects of type unsigned char (or, equivalently, std::byte) beginning at the same address as the T object.

The value representation of an object is the set of bits that hold the value of its type T.

For TriviallyCopyable types, value representation is a part of the object representation, which means that copying the bytes occupied by the object in the storage is sufficient to produce another object with the same value (except if the value is a trap representation[[1]](#footnote-1) of its type and loading it into the CPU raises a hardware exception, such as SNaN ("signalling not-a-number") floating-point values or NaT ("not-a-thing") integers).

The reverse is not necessarily true: two objects of TriviallyCopyable type with different object representations may represent the same value. For example, multiple floating-point bit patterns represent the same special value NaN. More commonly, some bits of the object representation may not participate in the value representation at all; such bits may be padding introduced to satisfy alignment requirements, bit field sizes, etc.

#include <cassert>

struct S {

char c; // 1 byte value

// 3 bytes padding

float f; // 4 bytes value

bool operator==(const S& arg) const { // value-based equality

return c == arg.c && f == arg.f;

}

};

assert(sizeof(S) == 8);

S s1 = {'a', 3.14};

S s2 = s1;

reinterpret\_cast<char\*>(&s1)[2] = 'b'; // change 2nd byte

assert(s1 == s2); // value did not change

For the objects of type char, signed char, and unsigned char (unless they are oversize bit fields), every bit of the object representation is required to participate in the value representation and each possible bit pattern represents a distinct value (no padding, trap bits, or multiple representations allowed).

1.5.1.2 Subobjects

An object can contain other objects, which are called subobjects. These include

* member objects
* base class subobjects
* array elements

An object that is not a subobject of another object is called complete object.

Complete objects, member objects, and array elements are also known as most derived objects, to distinguish them from base class subobjects. The size of a most derived object that is not a bit field is required to be non-zero (the size of a base class subobject may be zero: see empty base optimization).

Any two objects with overlapping lifetimes (that are not bit fields) are guaranteed to have different addresses unless one of them is a subobject of another or provides storage for another, or if they are subobjects of different type within the same complete object, and one of them is a zero-size base.

static const char c1 = 'x';

static const char c2 = 'x';

assert(&c1 != &c2); // same values, different addresses

1.5.1.3 多态对象

Objects of a class type that declares or inherits at least one virtual function are polymorphic objects. Within each polymorphic object, the implementation stores additional information (in every existing implementation, it is one pointer unless optimized out), which is used by virtual function calls and by the RTTI features (dynamic\_cast and typeid) to determine, at run time, the type with which the object was created, regardless of the expression it is used in.

For non-polymorphic objects, the interpretation of the value is determined from the expression in which the object is used, and is decided at compile time.

#include <iostream>

#include <typeinfo>

struct Base1 {

// polymorphic type: declares a virtual member

virtual ~Base1() {}

};

struct Derived1 : Base1 {

// polymorphic type: inherits a virtual member

};

struct Base2 {

// non-polymorphic type

};

struct Derived2 : Base2 {

// non-polymorphic type

};

int main()

{

Derived1 obj1; // object1 created with type Derived1

Derived2 obj2; // object2 created with type Derived2

Base1& b1 = obj1; // b1 refers to the object obj1

Base2& b2 = obj2; // b2 refers to the object obj2

std::cout << "Expression type of b1: " << typeid(decltype(b1)).name() << ' '

<< "Expression type of b2: " << typeid(decltype(b2)).name() << '\n'

<< "Object type of b1: " << typeid(b1).name() << ' '

<< "Object type of b2: " << typeid(b2).name() << '\n'

<< "size of b1: " << sizeof b1 << ' '

<< "size of b2: " << sizeof b2 << '\n';

}

Output:

Expression type of b1: Base1 Expression type of b2: Base2

Object type of b1: Derived1 Object type of b2: Base2

size of b1: 8 size of b2: 1

1.5.1.3 Strict aliasing

Accessing an object using an expression of a type other than the type with which it was created is undefined behavior in many cases, see reinterpret\_cast for the list of exceptions and examples.

1.5.1.4 Alignment

Every object type has the property called alignment requirement, which is an integer value (of type std::size\_t, always a power of 2) representing the number of bytes between successive addresses at which objects of this type can be allocated. The alignment requirement of a type can be queried with alignof or std::alignment\_of. The pointer alignment function std::align can be used to obtain a suitably-aligned pointer within some buffer, and std::aligned\_storage can be used to obtain suitably-aligned storage.

Each object type imposes its alignment requirement on every object of that type; stricter alignment (with larger alignment requirement) can be requested using alignas.

In order to satisfy alignment requirements of all non-static members of a class, padding may be inserted after some of its members.

#include <iostream>

// objects of type S can be allocated at any address

// because both S.a and S.b can be allocated at any address

struct S {

char a; // size: 1, alignment: 1

char b; // size: 1, alignment: 1

}; // size: 2, alignment: 1

// objects of type X must be allocated at 4-byte boundaries

// because X.n must be allocated at 4-byte boundaries

// because int's alignment requirement is (usually) 4

struct X {

int n; // size: 4, alignment: 4

char c; // size: 1, alignment: 1

// three bytes padding

}; // size: 8, alignment: 4

int main()

{

std::cout << "sizeof(S) = " << sizeof(S)

<< " alignof(S) = " << alignof(S) << '\n';

std::cout << "sizeof(X) = " << sizeof(X)

<< " alignof(X) = " << alignof(X) << '\n';

}

Output:

sizeof(S) = 2 alignof(S) = 1

sizeof(X) = 8 alignof(X) = 4

The weakest alignment (the smallest alignment requirement) is the alignment of char, signed char, and unsigned char, which equals 1; the largest fundamental alignment of any type is the alignment of std::max\_align\_t. If a type's alignment is made stricter (larger) than std::max\_align\_t using alignas, it is known as a type with extended alignment requirement. A type whose alignment is extended or a class type whose non-static data member has extended alignment is an over-aligned type. It is implementation-defined if new-expression, std::allocator::allocate, and std::get\_temporary\_buffer support over-aligned types. Allocators instantiated with over-aligned types are allowed to fail to instantiate at compile time, to throw std::bad\_alloc at runtime, to silently ignore unsupported alignment requirement, or to handle them correctly.

## **1.6 Definitions and ODR**

1.7 Name lookup

1.8 Qualified name lookup (qualified - unqualified)

A qualified name is a name that appears on the right hand side of the scope resolution operator :: (see also qualified identifiers). A qualified name may refer to a

* class member (including static and non-static functions, types, templates, etc)
* namespace member (including another namespace)
* enumerator

If there is nothing on the left hand side of the ::, the lookup considers only declarations made in the global namespace scope (or introduced into the global namespace by a using declaration). This makes it possible to refer to such names even if they were hidden by a local declaration:

#include <iostream>

int main() {

struct std{};

std::cout << "fail\n"; // Error: unqualified lookup for 'std' finds the struct

::std::cout << "ok\n"; // OK: ::std finds the namespace std

}

Before name lookup can be performed for the name on the right hand side of ::, lookup must be completed for the name on its left hand side (unless a decltype expression is used, or there is nothing on the left). This lookup, which may be qualified or unqualified, depending on whether there's another :: to the left of that name, considers only namespaces, class types, enumerations, and templates whose specializations are types:

struct A {

static int n;

};

int main() {

int A;

A::n = 42; // OK: unqualified lookup of A to the left of :: ignores the variable

A b; // Error: unqualified lookup of A finds the variable A

}

## 1.9 存储模型和数据竞争

Defines the semantics of computer memory storage for the purpose of the C++ abstract machine.

The memory available to a C++ program is one or more contiguous sequences of bytes. Each byte in memory has a unique address.

### **1.9.1 字节**

字节byte是内存中最小的可寻址单元。是由连续的位组成，足够保存任何UTF-8代码单元（256个不同的值）和基本执行字符集的任何成员（96个字符要求是单字节）。与C相似，C++支持大小为8位或者更大的字节。

The types char, unsigned char, and signed char use one byte for both storage and value representation. The number of bits in a byte is accessible as CHAR\_BIT or std::numeric\_limits<unsigned char>::digits.

类型char，unsigned char，signed char使用一个字节进行存储和值表达。一个字节中的

### **1.9.2 存储单元**

存储单元是：

* 标量类型的对象 (运算符类型，指针类型，枚举类型或std::nullptr\_t)
* 或者最大的非零的连续位域

注意：C++语言多个特征，诸如引用和虚函数（virtual），可能会引入程序不可访问的存储单元，而是由实现管理的。

struct S {

char a; // 存储位置 #1

int b : 5; // 存储位置 #2

int c : 11, // 存储位置 #2 (连续的)

: 0,

d : 8; // 存储位置 #3

struct {

int ee : 8; // 存储位置 #4

} e;

} obj; // 对象 'obj' 由独立的4个存储位置

### **1.9.3 线程和数据竞争**

通过std::thread::thread，std::async，或者其他方法调用顶层函数，实现线程的执行，其执行过程是一个流控过程。

在程序中，任何线程潜在地可以访问任何对象（带有automatic和thread-local存储周期的对象可以被其它线程使用指针或引用进行访问）。

不同的线程同时访问不同的存储单元，对其进行读写，没有干扰，也没有同步的要求。

当一个表达式的值写入一块存储单元的同时，另一表达式值也在读或者修改同一存储单元，这样就会发生冲突。有两个冲突的值的程序就会发生“数据竞争”除非

* 两个冲突的求值表达式在相同的线程或者同一个信号处理函数里执行，或
* 两个冲突的求值表达式都是原子操作（见：std::atomic），或
* 两个冲突的求值表达式执行是顺序的（见std::memory\_order）

如果发生数据竞争，程序的行为是未定义状态。

（可以使用std::mutex和另一个线程进行同步，来避免数据竞争问题）

int cnt = 0;

auto f = [&]{cnt++;};

std::thread t1{f}, t2{f}, t3{f}; // 未定义的行为

std::atomic<int> cnt{0};

auto f = [&]{cnt++;};

std::thread t1{f}, t2{f}, t3{f}; // OK

## **1.10 转译阶段**

编译器处理C++源文件的过程可以看做下面几个阶段：

### **1.10.1 阶段1**

1）源文件的每一个字节都会被映射为基本字符集中的字符。尤其是，依赖于操作系统的行结束符都会被换行符取代。基本字符集包含96个字符：

* 5 种空白字符 (空格， 水平制表符，垂直制表符，换页符，换行符)；
* 10 个数字[0-9]；
* 26 个英文字母的大小写；
* 29 标点符号：\_ { } [ ] # ( ) < > % : ; . ? \* + - / ^ & | ~ ! = , \ " '

2）源文件中，任何不能由基本字符集映射的字符，使用通用字符取代（即使用\转义符号进行转义）或者由其编译器作相应的处理。

3）三字符序列被相应的单字符表达方式替代。（until C++17）

### **1.10.2 阶段2**

1）一旦在某一行的结尾发现反斜杠“\”（后面紧跟换行符），符号“\”和换行符被删掉，把源文件的两行连接成一行。这是一次单程操作；如果一行结束时，后面紧跟两个反斜杠“\”字符和空行，那么是不会把这三行组成一个新行的。如果在这个阶段，出现通用字符（\uXXX），这种行为未被定义。

如果这个阶段之后，非空的源文件没有以换行符结束（不论是本来就没有换行符，或它以反斜杠符号结束），C++11之前没有定义这种行为，C++11之后会添加一个换行符。

### **1.10.3 阶段3**

1） 源文件被解析成注释，空白字符序列（空格，水平制表符，换行符，垂直制表符，和换页符），预处理符号，如下所示：

* 头文件名称，例如 <iostream> 或者 "myfile.h"
* 标识符
* 预处理数字
* 字符或字符串，包括用户自定义的 (C++11之后添加)
* 操作符和标点符号(包括可替换的符号), 如+，<<=，new，<%，##，and（&&）
* 不属于其它种类的单个非空白字符

2） 在阶段1和阶段2里，任何实施过转换的由双引号“”包含的原始字符串都会被恢复。（C++11之后）

3） 注释被一个空格替代

换行符被保留，没有明确说明，非换行符的空白字符序列是否会被整合成一个空格字符。

### **1.10.4 阶段4**

1）预处理程序被执行；

2）递归遍历阶段1到阶段4，用#include指令引入每一个文件；

3）在这个阶段结束时，所有源文件的预处理指令被移除。

### **1.10.5 阶段5**

1）由源文件转换而来的字符文字和字符串文字里的所有字符都被设为可执行字符集（有可能是如UTF-8一样的多字节字符集）

2） 字符文字和非原始字符串文字里的转义序列和通用字符被展开并转换为可执行字符集。如果由通用字符指定的字符不是可执行字符集里的成员，结果是编译器指定，但是保证不是一个null字符（广义上）

注意：这个阶段的转换执行，在某些编译器的实现里，可以由命令行选项进行控制：gcc和clang使用选项-finput-charset 指定源字符集的编码格式；-fexec-charset和-fwide-exec-charset，指定字符串和字符文字的可执行字符集的编码格式（没有编码前缀）。（C++11之后）

### **1.10.6 阶段6**

相邻字符串文字链接为新的字符串文字。

### **1.10.7 阶段7**

编译阶段：每个预处理符号被转换为一个符号。这些符号被从语法上和语义上分析，然后转换为一个翻译单元。

### **1.10.8 阶段8**

每一个翻译单元被检查，找出要求模板实例化的列表，包括哪些被要求显式实例化的模板。找到那些被要求实例化的模板，然后执行，产生实例化单元。

### **1.10.9 阶段9**

翻译单元，实例化单元，库组件满足外部引用的被集成都一个程序镜像，它包括在执行环境里执行时需要的信息。

### **1.10.10 注意**

一些编译器不实现实例化单元（也被称为模板仓库或模板注册表）且在阶段7编译每一个模板实例，然后存储代码到对应的目标文件中，然后链接器在阶段9把这些编译后的目标文件集合为一个可执行文件。

### **1.10.11 参考**

* C++11 standard (ISO/IEC 14882:2011):
* 2.2 Phases of translation [lex.phases]
* C++98 standard (ISO/IEC 14882:1998):
* 2.1 Phases of translation [lex.phases]

## 1.11 The main() function

# **5 声明**

## **5.7 存储周期和链接**

### **5.7.1 存储类关键字**

存储类关键字是名称声明语法的decl-specifier-seq的一部分。和名称的作用域一起，控制着名称的两个独立属性，自动存储期和链接属性。

* auto 自动存储期。（C++11之前适用）
* register 自动存储期。另外，提醒编译器把对象放入处理器的寄存器中。（C++17之前适用，现已被废弃）
* static 静态或线程存储期，内部链接属性。
* extern 静态或线程存储期，外部链接属性。
* thread\_local 线程存储期。（C++11之后适用）

一次只能一个存储类关键字出现在声明语句中，thread\_local是个例外，需要与static和或者extern结合使用。（C++11之后适用）

### **5.7.2 解释**

* 关键字auto只能声明在块作用域或函数参数列表中的对象。它代表着其默认是自动存储期。在C++11中，这个关键字的意义被改变。
* 关键字register也只能声明在块作用域或函数参数列表中的对象。 默认是自动存储期。另外，这个关键字提示代码优化器保存该变量的值在CPU寄存器里。C++11放弃了这个关键字。
* 关键字static，能够在对象的声明（除函数列表外），函数的声明（除在块作用域）和不具名联合体声明里使用。当用在类成员上时，它声明了一个静态成员。当用在对象声明上时，它指定了静态存储期（如果和thread\_local联合使用除外）。当用在命名空间作用域内时，它指定了内部链接属性。
* 关键字extern只被允许用在变量和函数的声明上（除了类成员或函数参数）。它指定了外部链接属性，且不会影响存储期，但是它不能被用在一个具有自动存储期的对象身上，所以，所有的extern对象具有static或thread存储周期。另外，使用了extern且没有初始化的变量声明不是一个定义。
* 关键字thread\_local被允许声明命名空间范围和块作用域内的对象，及静态数据成员。它指明，对象具有线程存储周期。可以和static或extern关键字一起使用，指明内部或者外部链接属性（除了static数据成员，其余的总是具有外部链接属性），但是添加的static关键字不会影响其存储周期。

### **5.7.3 存储期**

所有的对象都具有下面这些存储类型中的一种：

* automatic

对象在代码块开始时被分配，离开时收回分配的存储空间。所有的局部对象都有这种存储周期，除非，它们被声明为static，extern或thread\_local。

* static

当程序开始运行时分配对象的存储空间，程序结束时收回对象的存储空间。只允许一个对象实例存在。所有声明在命名空间的对象（包括全局命名空间），前面加上static，或者extern关键字的都有这种存储周期。

* thread

线程开始分配对象，线程结束收回分配给对象的存储空间。每个线程拥有这个对象唯一的实例。只有使用关键字thread\_local声明的对象才有这种存储周期。关键字thread\_local 可以和static或extern结合使用，以调整链接属性。

* dynamic

当使用动态内存分配函数请求分配或者回收对象的存储空间时才会使用。

### **5.7.4 链接属性**

名称泛指对象，引用，函数，类型，模板，命名空间，或数值（枚举器），都可以有链接属性。如果一个名称具有链接属性，那么在另一个作用域内声明而引入的相同名称就会被引用为同一个实体。如果在几个作用域内声明了具有相同名称的变量，函数，或另一个实体，但是又没有有效的链接属性，那么就会产生几个实体实例。

链接类型可以被分为下面三种：

（1）无链接。

这种方式适用于名称在它的作用域内的情况。下面的几种情况具有非链接属性：

* 名称没有显式地使用extern关键字声明（无关static修饰符）；
* 局部类和它的成员函数；
* 块作用域内声明的其它名称，例如typedef，enum等声明的名称；

（2）内部链接。

在当前的编译单元里能够被所有的作用域引用的名称。命名空间作用范围内声明的下面中的任何一种名称都具有内部链接属性：

* static声明的变量，函数，和函数模板；
* 没有使用extern声明或者之前也没有被声明为具有外部链接属性的非易失性（non-volatile）非内嵌常量限定的变量（包括constexpr）；
* 不具名联合体的数据成员；

另外，在不具名命名空间或者不具名命名空间内部的命名空间里声明的所有变量，即使明确使用extern声明，也是内部链接属性。

（3）外部链接

可被其它编译单元参考引用的名称，就具有外部链接属性。具有外部链接属性的变量和函数也具有语言链接属性，这使得链接不同程序语言写的编译单元成为可能。

任何在命名空间里声明的下列变量都具有外部链接属性，除非，命名空间是不具名的或者被一个不具名命名空间包含（C++11之后）。

* 上面没有列出的变量和函数（也就是说，没有被声明为static函数，命名空间范围内的非const变量没有被声明为static，任何声明为extern的变量）
* 枚举和枚举器
* 类名，它们的成员函数，静态数据成员（const与否），嵌套类和枚举类型变量，类体内首次使用友邻声明的函数
* 上面没有列出的所有模板变量（就是说，声明为static的非函数模板）

首次声明在块作用域内的下列变量中任何一种都有外部链接属性：

* 声明为extern的变量
* 函数变量

### **5.7.5 静态局部变量**

使用限定符static声明在块作用域内的变量具有static存储期，只有当第一次执行经过它们的声明时被初始化（除非它们的初始化是0或常量初始化，这种初始化可以在进入块作用域之前就已经完成）。在所有后面的调用中，声明都会被跳过，不执行。

如果初始化过程出现异常，那么不认为变量被初始化，再一次尝试控制经过声明语句时，还会初始化。

如果多个线程同时尝试初始化相同的静态变量，初始化也只会进行一次（对于使用std::cal\_once的任意函数都能获得相似的行为）。

注意：这个功能的通常实现就是使用双重检查锁定模式，它可以减少已经初始化为局部静态和单个非原子boolean的比较产生的系统开销。

当程序exit时，调用块作用域的析构函数，但前提是初始化成功。

对于同一个内嵌函数（也许是隐含内嵌）的所有定义里的局部静态对象，都会被一个编译单元定义的相同的对象引用。

### **5.7.6 注意**

在C语言中，在顶层命名空间作用域（相当于C的文件范围）内的，是const且没有extern修饰的名称具有外部属性，但是在C++中却是内部链接属性。

在C里，寄存器变量的地址不能获取，但是在C++中，变量声明为register和没有任何存储类关键字修饰是没有什么区别的。（C++11之前）

C++中，不像C，变量不能声明为register。（C++17之后）

具有内部或外部链接属性的thread\_local型变量的名称可以被不同的实例引用，依赖于代码是否在同一个或者不同的线程中执行。

关键字extern也可以指定语言链接属性和明确的模板实例声明，但是它不是存储类限定符（除非，声明被直接包含在语言链接指定中，在这种情况时，声明被像包含extern限定符一样对待）。

在C++语法中，关键字mutable是存储类限定符，尽管它不会影响存储周期或链接属性。

现在这段是不完整的，因为在同一个编译单元重新声明的规则。

存储类限定符，对于thread\_local是个例外，不允许明确的指定和明确的实例。

template <class T> struct S {

thread\_local static int tlm;

};

template <> thread\_local int S<float>::tlm = 0; // "static" 没有出现在这里

### **5.7.7 关键字**

auto, register, static, extern, thread\_local

### **5.7.8 举例说明**

#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; // 锁外修改是没问题的;这是一个thread-local 变量

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();

}

可能的输出：

Rage counter for a: 2

Rage counter for main: 1

Rage counter for b: 2

# **10 Templates**

A template is a C++ entity that defines one of the following:

* a family of classes (class template), which may be nested classes
* a family of functions (function template), which may be member functions
* an alias to a family of types (alias template) (since C++11)
* a family of variables (variable template) (since C++14)
* a concept (constraints and concepts) (since C++20)

## **10.9 Parameter pack**

A template parameter pack is a template parameter that accepts zero or more template arguments (non-types, types, or templates). A function parameter pack is a function parameter that accepts zero or more function arguments.

A template with at least one parameter pack is called a variadic template.

1. trap representation: 缺陷位 [↑](#footnote-ref-1)