# 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)
```

- （1）常被称为"C-风格"或多行注释。
- （2）常被称为"C++-风格"或单行注释。

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 名称

名称包含以下几种情况，每一种情况必与一个实体或者标签相关联：

- 标识符；
- 函数表达式里的重载运算符的名称 (例如 operator+, operator new)；
- 用户自定义的类型转换函数 (operator bool)；
- 用户定义的文字操作符名称(operator "" _km)；
- 后跟参数列表的模板名称 (MyTemplate<int>)。

程序中每一个能够表示实体（entity）的名称，都是通过声明引入程序的。表示标签的每一个名称都是通过 goto 语句或者标签语句引入的。在多个翻译单元中使用的名称可以指相同或不同的实体，这取决于链接。

当编译器在程序中遇到一个未知的名字时，除了模板声明和定义中的依赖名之外，它将该未知的名称与通过名称查找引入的声明关联起来。（对于这些名称，由编译器确定它们是类型，模板，还是其它实体，这可能需要严格的消除歧义）。

**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.

**Comment [SW1]:** 函数模板定制

### 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.

Comment [SW2]: 泛左值

```
// 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.

Comment [SW3]: 纯右值

### 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 | at least 64 | 64 | 64 | 64 | 64 |

| | | | | | | |
|---|---|---|---|---|---|---|
| long long int | (C++11) | | | | | |
| 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.

**Comment [SW4]:** 极限情况，极端情况

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

**Comment [SW5]:** 值表示

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] 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).

**Comment [SW6]:** 浅拷贝

**Comment [SW7]:** NAN：主角来了，如果指数全部为 1，尾数非 0，则表示这个值不是一个真正的值（Not A Number)。NAN 又分成两类：QNAN（Quiet NAN）和 SNAN（Singaling NAN)。QNAN 与 SNAN 的不同之处在于，QNAN 的尾数部分最高位定义为 1，SNAN 最高位定义为 0；QNAN 一般表示未定义的算术运算结果，最常见的莫过于除 0 运算；SNAN 一般被用于标记未初始化的值，以此来捕获异常

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.

[1] trap representation：缺陷位

```
#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 名称查找

名称查找就是把程序中遇到的名称与引入它的声明结合的过程。

例如，为了编译 std::cout << std::endl;，编译器执行如下步骤：

- 为名称 std 进行非限定名称查找，在头文件<iostream>中发现命名空间 std 的声明
- 对 cout 进行限定名称查找，在命名空间 std 中发现变量声明
- 对 endl 进行限定名称查找，在命名空间 std 中发现函数模板声明
- 对命名空间 std 中具有多个函数模板声明的名称 operator<<进行参数依赖查找，对类 std::ostream 中的多个成员函数声明进行限定名称查找

For function and function template names, name lookup can associate multiple declarations with the same name, and may obtain additional declarations from argument-dependent lookup. Template argument deduction may also apply, and the set of declarations is passed to overload resolution, which selects the declaration that will be used. Member access rules, if applicable, are considered only after name lookup and overload resolution.

For all other names (variables, namespaces, classes, etc), name lookup must produce a single declaration in order for the program to compile. Lookup for a name in a scope finds all declarations of that name, with one exception, known as the "struct hack" or "type/non-type hiding": Within the same scope, some occurrences of a name may refer to a declaration of a class/struct/union/enum that is not a typedef, while all other occurrences of the same name either all refer to the same variable, non-static data member (since C++14), or enumerator, or they all refer to possibly overloaded function or function template names. In this case, there is no error, but the type name is hidden from lookup (the code must use elaborated type specifier to access it).

对于函数和函数模板名称，名称查找可以将多个声明和相同的名称进行关联，且可通过参数依赖查找获得额外的声明。应用模板参数推导，将这些声明传递给重载决策，由它选择使用哪个声明。成员访问规则，只有在名称查找和重载决策之后才会被考虑。

### 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.

C++程序可用的内存是一个或多个连续的字节序列。内存中的每个字节都有一个唯一的地址。

### 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
```

1.9.4 memory_order

When a thread reads a value from a memory location, it may see the initial value, the value written in the same thread, or the value written in another thread. See std::memory_order for details on the order in which writes made from threads become visible to other threads.

Forward progress

Obstruction freedom

When only one thread that is not blocked in a standard library function executes an atomic function that is lock-free, that execution is guaranteed to complete (all standard library lock-free operations are obstruction-free)

只有当一个调用标准库函数不会发生阻塞的线程，执行原子操作，这个执行才能够被保证完成。（所有的标准库的无锁操作都是非阻塞的）

Lock freedom

When one or more lock-free atomic functions run concurrently, at least one of them is guaranteed to complete (all standard library lock-free operations are lock-free -- it is the job of the implementation to ensure they cannot be live-locked indefinitely by other threads, such as by continuously stealing the cache line)

当一个或多个无锁的原子操作函数同时运行时，它们中至少一个被保证执行（所有标准库的无锁操作都是无锁的—这是由实现保证的，）

Progress guarantee

In a valid C++ program, every thread eventually does one of the following:

terminate

makes a call to an I/O library function

performs an access through a volatile glvalue

performs an atomic operation or a synchronization operation

终止

调用 IO 库函数

访问易失性 glvalue

执行原子操作或同步操作

No thread of execution can execute forever without performing any of these observable behaviors.

Note that it means that a program with endless recursion or endless loop (whether implemented as a for-statement or by looping goto or otherwise) has undefined behavior. This allows the compilers to remove all loops that have no observable behavior, without having to prove that they would eventually terminate.

A thread is said to make progress if it performs one of the execution steps above (I/O, volatile, atomic, or synchronization), blocks in a standard library function, or calls an atomic lock-free function that does not complete because of a non-blocked concurrent thread.

Concurrent forward progress

If a thread offers concurrent forward progress guarantee, it will make progress (as defined above) in finite amount of time, for as long as it has not terminated, regardless of whether other threads (if any) are making progress.

The standard encourages, but doesn't require that the main thread and the threads started by std::thread offer concurrent forward progress guarantee.

Parallel forward progress

If a thread offers parallel forward progress guarantee, the implementation is not required to ensure that the thread will eventually make progress if it has not yet executed any execution step (I/O, volatile, atomic, or synchronization), but once this thread has executed a step, it provides concurrent forward progress guarantees (this rule describes a thread in a thread pool that executes tasks in arbitrary order)

Weakly parallel forward progress

If a thread offers weakly parallel forward progress guarantee, it does not guarantee to eventually make progress, regardless of whether other threads make progress or not.

Such threads can still be guaranteed to make progress by blocking with forward progress guarantee delegation: if a thread P blocks in this manner on the completion of a set of threads S, then at least one thread in S will offer a forward progress guarantee that is same or stronger than P. Once that thread completes, another thread in S will be similarly strengthened. Once the set is empty, P will unblock.

The parallel algorithms from the C++ standard library block with forward progress delegation on the completion of an unspecified set of library-managed threads.

## 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 main() 函数

Ddd

## 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
```

**5.10 const/volatile – 常量表达式**

5.10.1 const 和 volatile 类型限定符

出现在任何类型说明符中，包括声明语法的 decl-specifier-seq，可以用来指定声明对象或者命名类型的常量性（constness）或者易变性（volatile）。

（1）const　　定义类型为常量；

（2）volatile　 定义类型为 volatile；

（3）mutable　 适用于非引用非常量的非静态类成员，并指定该成员不影响类的外部可见状态（如常用的 mutex，内存缓存，lazy evaluation 和 access instrumentation）。Const 类实例的 mutable 成员是可以修改的。（注意：C++语法虽然把 mutable 作为 storage-class-specifier，但是它并不影响存储类）。

5.10.2 解释

For any type T (including incomplete types), other than function type or reference type, there are three more distinct types in the C++ type system: *const-qualified* T, *volatile-qualified* T, and *const-volatile-qualified* T.

Note: array types are considered to have the same cv-qualification as their element types.

When an object is first created, the cv-qualifiers used (which could be part of *decl-specifier-seq* or part of a *declarator*in a declaration, or part of *type-id* in a new-expression) determine the constness or volatility of the object, as follows:

- ***const object*** - an object whose type is *const-qualified*, or a non-mutable subobject of a const object. Such object cannot be modified: attempt to do so directly is a compile-time error, and attempt to do so indirectly (e.g., by modifying the const object through a reference or pointer to non-const type) results in undefined behavior.
- ***volatile object*** - an object whose type is *volatile-qualified*, or a subobject of a volatile object, or a mutable subobject of a const-volatile object. Every access (read or write operation, member function call, etc.) made through a glvalue expression of volatile-qualified type is treated as a visible side-effect for the purposes of optimization (that is, within a single thread of execution, volatile accesses cannot be optimized out or reordered with another visible side effect that is sequenced-before or sequenced-after the volatile access. This makes volatile objects suitable for communication with a signal handler, but not with another thread of execution, see std::memory_order). Any attempt to refer to a volatile object through a non-volatile glvalue (e.g. through a reference or pointer to non-volatile type) results in undefined behavior.
- ***const volatile object*** - an object whose type is *const-volatile-qualified*, a non-mutable subobject of a const volatile object, a const subobject of a volatile object, or a non-mutable volatile subobject of a const object. Behaves as both a const object and as a volatile object.

对于任何类型 T（包括不完整的类型），除了函数类型或引用类型，C++类型系统还有三种不同的类型：const 限定类型，volatile 限定类型，常量 volatile 限定类型

## 7 函数

7.1 函数声明

7.2 默认参数

7.3 可变参数

### 7.4 Lambda 表达式(C++11)

构造闭包：一个未命名的函数对象，能够捕获范围内的变量。

7.4.1. 语法讲解

1 语法

[ captures ] <tparams>(optional)(c++20) (params) specifiers exception attr -> ret requires(optional) (c++20) { body }
（1）

[ captures ] ( params ) -> ret { body }　（2）

[ captures ] ( params ) { body }　（3）

[ captures ] { body }　（4）

（1）第一种情况是完整声明；

（2）声明一个常量 lambda 表达式：通过复制捕获的对象在 lambda 体中是 const 的。

（3）省略拖尾返回类型：闭包的操作符（）的返回类型是由下面规则决定的：

| |
|---|
| 1. 如果主体只包含带表达式的单个 return 语句，则返回类型是 return 表达式的类型（在左值到右值，数组到指针或函数到指针的隐式转换之后）。(until C++14) |
| 2. 否则，返回类型是 void。(until C++14) |
| 返回类型是从 return 语句中推导出来的，仿佛用 auto 声明返回类型的函数一样。（C++14 后） |

（4）省略参数列表：函数不带任何参数，就像参数列表是()一样。这种形式，只有在没有 constexpr，mutable，异常说明，属性或拖尾返回类型的情况下才会使用。

2 解释

（1）captures

使用逗号分割的 0 或者多个捕获的列表，可以选择使用默认捕获开始。捕获列表通过以下方法进行传递（详细的介绍可以查看下面的描述）：

- [a, &b]  a 通过复制捕获，b 通过引用捕获
- [this]  通过引用捕获当前对象
- [&]  通过引用捕获所有 lambda 表达式里使用的自动存储的变量，以及捕获当前对象，如果存在
- [=]  通过复制捕获所有 lambda 表达式里使用的自动存储的变量，以及捕获当前对象，如果存在
- []  什么也不捕获

Lambda 表达式也可以不通过捕获而使用一个变量，但是该变量必须具有：

- 是非局部变量，有 static 或 thread local 存储周期（在这种情况下，变量不能够被捕获）；
- 是使用常量表达式进行了初始化的引用。

Lambda 表达式可以读取变量的值，不用捕获，但必须具有如下条件：

- 是 const non-volatile 整形或枚举型且使用常量表达式进行初始化
- 常量表达式且不能复制构造的。

| | |
|---|---|
| Structured bindings 不能被捕获 | C++17 后 |

（2）<tparams> （C++20）

略。

（3）params

参数列表，像命名函数那样，C++14 之前，默认参数不被允许。如果 auto 作为一个参数的类型，lambda 是一个通用 lambda。（C++14 之后）

（4）specifiers

说明符的可选序列，允许使用一下说明符:

- mutable：允许在函数体内修改通过复制捕获的参数，且调用它们的 non-const 成员函数。
- constexpr：显式地指定函数调用操作符是一个 constexpr 函数。当这个说明服不存在时，如果恰好满足所有的 constexpr 函数要求，函数调用操作符总是 constexpr。

（5）exception

provides the exception specification or the noexcept clause for operator() of the closure type

（6）attr

provides the attribute specification for operator() of the closure type

（7）ret

返回类型。如果不存在，就由函数 return 语句隐含指定（或者返回 void 类型）。

（8）requires

为闭包类型的 operator()操作添加约束条件。

（9）body

函数体

Lambda 表达式是一个纯右值表达式，它既不是 union 也不是 aggregate（集合），就是一个未命名的 class 类型，称之为闭包类型。在最小的块作用域，类作用域或包含 lambda 的命名空间作用域中被声明（用于 ADL） 表达。封闭类型有以下成员：

ClosureType::operator()(params)

| | |
|---|---|
| ret operator() (params) const {body} | 关键字 mutable 没有被使用 |
| ret operator() (params) {body} | 关键字 mutable 被使用 |
| template<template-params><br>ret operator()(params) const { body } | （C++14 之后）<br>通用 Lambda 表达式 |
| template<template-params><br>ret operator()(params) { body } | （C++14 之后）<br>通用 Lambda 表达式，关键字 mutable 被使用 |

当被调用时，执行 lambda 表达式的主体。当访问变量时，访问它捕获的复制变量（通过复制捕获的实体），或原始对象（通过引用捕获的实体）。除非在 lambda 表达式中使用关键字 mutable，否则 const 限定的函数调用操作符和通过复制捕获的对象，在 operator()内部是不可修改的。函数调用操作符永远不会是 volatile 限定或 virtual 限定的。

如果满足常量表达式函数的要求，函数调用操作符总是 constexpr 的。如果在 Lambda 表达式中使用了 constexpr，那么它也是 constexpr 的。

对于在 params 中指定的每一个参数，它的类型被指定为 auto，那么，按照参数的先后顺序，一个虚构的模板参数被添加到 template-params 中。如果 params 的相应函数成员是函数参数包，那么这个虚构的模板参数可能是一个参数包。

```
    // generic lambda, operator() is a template with two parameters
    auto glambda = [](auto a, auto&& b) { return a < b; };
    bool b = glambda(4, 3.14); // ok

    // generic lambda, operator() is a template with one parameter
    auto vglambda = [](auto printer)
    {
        return [=](auto&&... ts) //  通用 Lambda 表达式, ts 参数包
        {
            printer(std::forward<decltype(ts)>(ts)...);
            return [=] { printer(ts...); }; // nullary lambda (takes no parameters)
        };
    };
    auto p = vglambda([](auto v1, auto v2, auto v3)
        { std::cout << v1 << v2 << v3; });
    auto q = p(1, 'a', 3.14); // outputs 1a3.14
    q();                              // outputs 1a3.14
```

ClosureType's operator() cannot be explicitly instantiated or explicitly specialized.

If the lambda definition uses an explicit template parameter list, that template parameter list is used with operator(). For every parameter in params whose type is specified as auto, an additional invented template parameter is appended to the end of that template parameter list:

```
// generic lambda, operator() is a template with two parameters
auto glambda = []<class T>(T a, auto&& b) { return a < b; };

// generic lambda, operator() is a template with one parameter pack
auto f = []<typename ...Ts>(Ts&& ...ts) {
    return foo(std::forward<Ts>(ts)...);
};
```

（C++20 之后）

the exception specification *exception* on the lambda-expression applies to the function-call operator or operator template.

For the purpose of name lookup, determining the type and value of the this pointer and for accessing non-static class members, the body of the closure type's function call operator is considered in the context of the lambda-expression.

```
struct X {
    int x, y;
    int operator()(int);
    void f()
```

```
        {
                // the context of the following lambda is the member function X::f
                [=]()->int
                {
                        return operator()(this->x + y); // X::operator()(this->x + (*this).y)
                                                                          // this has type X*
                };
        }
};
```

ClosureType's operator() cannot be named in a friend declaration.

**Dangling references**

If a non-reference entity is captured by reference, implicitly or explicitly, and the function call operator of the closure object is invoked after the entity's lifetime has ended, undefined behavior occurs. The C++ closures do not extend the lifetimes of the captured references.

Same applies to the lifetime of the object pointed to by the captured this pointer.

| | |
|---|---|
| using F = ret(*)(params);<br>operator F() const; | (capture-less non-generic lambda) |
| using F = ret(*)(params);<br>constexpr operator F() const; | (capture-less non-generic lambda) |
| template<template-params> using fptr_t = /*see below*/;<br>template<template-params> operator fptr_t<template-params>() const; | (capture-less generic lambda) |
| template<template-params> using fptr_t = /*see below*/;<br>template<template-params> operator fptr_t<template-params>() const; | (capture-less generic lambda) |

This user-defined conversion function is only defined if the capture list of the lambda-expression is empty. It is a public, constexpr (since C++17) non-virtual, non-explicit, const noexcept (since C++14) member function of the closure object.

| | |
|---|---|
| A generic captureless lambda has user-defined conversion function template with the same invented template parameter list as the function-call operator template. If the return type is empty or auto, it is obtained by return type deduction on the function template specialization, which, in turn, is obtained by template argument deduction for conversion function templates. | （C++14 之后） |
| void f1(int (*)(int)) {}<br>void f2(char (*)(int)) {} | |

| | |
|---|---|
| void h(int (*)(int)) {} // #1<br><br>void h(char (*)(int)) {} // #2<br><br>auto glambda = [](auto a) { return a; };<br><br>f1(glambda); // ok<br><br>f2(glambda); // error: not convertible<br><br>h(glambda); // ok: calls #1 since #2 is not convertible<br><br><br>int& (*fpi)(int*) = [](auto* a)->auto& { return *a; }; // ok | |

The value returned by this conversion function is a pointer to a function with C++ language linkage that, when invoked, has the same effect as invoking the closure object's function call operator directly.

| | |
|---|---|
| This function is constexpr if the function call operator (or specialization, for generic lambdas) is constexpr.<br><br>auto Fwd= [](int(*fp)(int), auto a){return fp(a);};<br>auto C=[](auto a){return a;};<br>static_assert(Fwd(C,3)==3);//OK<br>auto NC=[](auto a){static int s; return a;};<br>static_assert(Fwd(NC,3)==3); // error: no specialization can be constexpr because of s<br><br>If the closure object's operator() has a non-throwing exception specification, then the pointer returned by this function has the type pointer to noexcept function. | （C++17 之后） |

3 Lambda 捕获

字段 captures 是用逗号分隔的 0 或多个捕获的列表，可以选择使用默认捕获。唯一的默认捕获是

- & 通过引用隐式捕获 odr-used[2]自动存储变量
- = 通过复制隐式捕获 odr-used 自动变量

捕获的语法：

| | |
|---|---|
| identifier | 复制捕获 |
| identifier...... | 复制捕获，标识符列表 |

[2] 非正式得说，如果一个对象，它的值可以使用它的地址或者引用进行读（除非是编译时常量）或写，那么它是 odr-used 的。如果一个引用，它被使用且引用体在编译的时候不被知道，那么它也是 odr-used 的。如果一个函数，被调用或者取其地址，那么它也是 odr-used 的。如果对象、引用或函数是 odr-used，那么程序中的某一处必定存在它的定义；如果违反，将会发生一个链接时错误。

| | |
|---|---|
| identifier　initializer | 使用初始化器的复制捕获 |
| & identifier | 引用捕获 |
| & identifier… | 引用捕获，标识符列表 |
| & identifier　initializer | 使用初始化器的引用捕获 |
| this | 当前对象的引用捕获 |
| *this | 当前对象的复制捕获 |

如果默认捕获是引用，那么后面的单个捕获就不能用引用&开始。

```
struct S2 { void f(int i); };
void S2::f(int i)
{
    [&]{};              // OK: 默认的引用捕获
    [&, i]{};           // OK: 引用捕获，i 是复制捕获
    [&, &i] {};         // Error: 默认捕获是引用时，后面的捕获不能再使用引用。
    [&, this] {};       // OK, 等于[&]
    [&, this, i]{};     // OK, 等于[&, i]
}
```

但是，如果默认捕获是=，那么后面单个捕获必须是&开始，或*this（C++17 后），或 this（C++20 后）。

```
struct S2 { void f(int i); };
void S2::f(int i)
{
    [=]{};              // OK: 默认复制捕获
    [=, &i]{};          // OK: 默认复制捕获，i 是引用捕获
    [=, *this]{};       // until C++17: Error: invalid syntax
                        // since c++17: OK: captures the enclosing S2 by copy
    [=, this] {};       // until C++20: Error: this when = is the default
                        // since C++20: OK, same as [=]
}
```

任何捕获只能出现一次：

```
struct S2 { void f(int i); };
void S2::f(int i)
```

```
{
    [i, i] {};              // Error: i  重复
    [this, *this] {};      // Error: "this"  重复  (C++17)
}
```

只有在块作用域或默认成员初始化器中定义的 lambda 表达式，才会有默认捕获或者没有初始化的捕获。对于这样的 lambda 表达式，作用范围被定义为包括最内层嵌套的函数及其参数。

在没有初始化的情况下（除了 this-capture），任何捕获中的标识符都会在 lambda 范围内使用通常的非限定名称查找来查找。查找的结果必须是在作用范围内声明的具有自动存储周期的变量。变量（或 this）被明确地捕获。

| | |
|---|---|
| 使用初始化器的捕获，表现行为就像声明并显式捕获一个声明类型为 auto 的变量，它的声明域是 lambda 表达式的主体（也就是说，它不在初始化器的范围之内），除了：<br>如果捕获是通过复制实现，闭包对象的非静态数据成员就是引用那个自动变量的另一种方法；<br>如果捕获是通过引用，那么当闭包对象的生命周期结束时，引用变量的生命周期也就结束了。<br>这常用来捕获如 `x = std::move(x)move-only` 类型<br>`int x = 4;`<br>`auto y = [&r = x, x = x + 1]()->int`<br>`    {`<br>`        r += 2;`<br>`        return x * x;`<br>`    }(); // updates ::x to 6 and initializes y to 25.` | C++14 后 |

7.5 inline 内嵌函数

## 7.6 参数依赖查找（ADL）

### 1. 举例说明

参数依赖检查，又称为 ADL（arguments dependent lookup）或 Koenig 查找，是在函数调用表达式中，也包括对重载运算符的隐式函数调用中，查找非限定函数名称的一组规则。除了通常的非限定名称查找所考虑的作用域和命名空间之外，这些函数名称还会被在其参数的命名空间中查找。

参数依赖检查使得使用定义在不同的命名空间里的操作符成为可能。例如：

```
#include <iostream>
int main()
    // 在全局命名空间中没有操作符<<，但是 ADL 检查 std 命名空间，因为左边的参数在 std 中，且在
    // 其中能找到 std::operator<<(std::ostream&, const char*)
    std::cout << "Test\n";
    // 相同，使用函数调用符号样式
    operator<<(std::cout, "Test\n");

    // 但是,下面这句就会报出"Error: 'endl' is not declared in this namespace"这样的错误
    // 因为这对于 endl()来说，并不是一个函数调用，所以 ADL 并不适用
    std::cout << endl;

    // OK：这是一个函数调用，ADL 检查 std 命名空间，因为 endl 的参数在 std 命名空间里,能够发现 std::endl
    endl(std::cout);

    // 但是，下面这句就会报出"Error: 'endl' is not declared in this namespace"这样的错误
    // 因为子表达式(endl)不是一个函数调用表达式
    (endl)(std::cout);
}
```

**2 详细介绍**

First, the argument-dependent lookup is not considered if the lookup set produced by usual unqualified lookup contains any of the following:

1) 类成员声明

2) a declaration of a function at block scope (that's not a using-declaration)

3) any declaration that is not a function or a function template (e.g. a function object or another variable whose name conflicts with the name of the function that's being looked up)

Otherwise, for every argument in a function call expression its type is examined to determine the *associated set of namespaces and classes* that it will add to the lookup.

1) For arguments of fundamental type, the associated set of namespaces and classes is empty

2) For arguments of class type (including union), the set consists of

a) The class itself

b) All of its direct and indirect base classes

c) If the class is a member of another class, the class of which it is a member

d) The innermost enclosing namespaces in the classes added to the set

3) For arguments whose type is a class template specialization, in addition to the class rules, the following types are examined and their associated classes and namespaces are added to the set

a) The types of all template arguments provided for type template parameters (skipping non-type template parameters and skipping template template parameters)

b) The namespaces in which any template template arguments are members

c) The classes in which any template template arguments are members (if they happen to be class member templates)

4) For arguments of enumeration type, the namespace in which the enumeration is defined is added to the set. If the enumeration type is a member of a class, that class is added to the set.

5) For arguments of type pointer to T or pointer to an array of T, the type T is examined and its associated set of classes and namespaces is added to the set.

6) For arguments of function type, the function parameter types and the function return type are examined and their associated set of classes and namespaces are added to the set.

7) For arguments of type pointer to member function F of class X, the function parameter types, the function return type, and the class X are examined and their associated set of classes and namespaces are added to the set.

8) For arguments of type pointer to data member T of class X, the member type and the type X are both examined and their associated set of classes and namespaces are added to the set.

9) If the argument is the name or the address-of expression for a set of overloaded functions (or function templates), every function in the overload set is examined and its associated set of classes and namespaces is added to the set.

a) Additionally, if the set of overloads is named by a template-id (template name with template arguments), all of its type template arguments and template template arguments (but not non-type template arguments) are examined and their associated set of classes and namespaces are added to the set.

If any namespace in the associated set of classes and namespaces is an inline namespace, its enclosing namespace is also added to the set.

If any namespace in the associated set of classes and namespaces directly contains an inline namespace, that inline namespace is added to the set.

After the associated set of classes and namespaces is determined, all declarations found in classes of this set are discarded for the purpose of further ADL processing, except namespace-scoped friend functions and function templates, as stated in point 2 below .

The set of declarations found by ordinary unqualified lookup and the set of declarations found in all elements of the associated set produced by ADL, are merged, with the following special rules

1) using-directives in the associated namespaces are ignored

2) namespace-scoped friend functions (and function templates) that are declared in an associated class are visible through ADL even if they are not visible through ordinary lookup

3) all names except for the functions and function templates are ignored (no collision with variables)

**Notes**

Because of argument-dependent lookup, non-member functions and non-member operators defined in the same namespace as a class are considered part of the public interface of that class (if they are found through ADL) [1].

ADL is the reason behind the established idiom for swapping two objects in generic code:

```
using std::swap;

swap(obj1, obj2);
```

because calling std::swap(obj1, obj2) directly would not consider the user-defined swap() functions that could be defined in the same namespace as the types of obj1 or obj2, and just calling the unqualified swap(obj1, obj2)would call nothing if no user-defined overload was provided. In particular, std::iter_swap and all other standard library algorithms use this approach when dealing with Swappable types.

Name lookup rules make it impractical to declare operators in global or user-defined namespace that operate on types from the std namespace, e.g. a custom operator>> or operator+ for std::vector or for std::pair (unless the element types of the vector/pair are user-defined types, which would add their namespace to ADL). Such operators would not be looked up from template instantiations, such as the standard library algorithms. See dependent namesfor further details.

ADL can find a friend function (typically, an overloaded operator) that is defined entirely within a class or class template, even if it was never declared at namespace level.

```
template<typename T>
struct number
{
        number(int);
        friend number gcd(number x, number y) { return 0; }; // definition within
                                                                                                    // a class template
};
// unless a matching declaration is provided gcd is an invisible (except through ADL)
// member of this namespace
void g() {
        number<double> a(3), b(4);
        a = gcd(a,b); // finds gcd because number<double> is an associated class,
                                    // making gcd visible in its namespace (global scope)
//    b = gcd(3,4); // Error; gcd is not visible
}
```

Although a function call can be resolved through ADL even if ordinary lookup finds nothing, a function call to a function template with explicitly-specified template arguments requires that there is a declaration of the template found by ordinary lookup (otherwise, it is a syntax error to encounter an unknown name followed by a less-than character)

```
namespace N1 {
    struct S {};
    template<int X> void f(S);
}
namespace N2 {
    template<class T> void f(T t);
}
void g(N1::S s) {
    f<3>(s);            // Syntax error (unqualified lookup finds no f)
    N1::f<3>(s);    // OK, qualified lookup finds the template 'f'
    N2::f<3>(s);    // Error: N2::f does not take a non-type parameter
                               //                N1::f is not looked up because ADL only works
                               //                            with unqualified names
    using N2::f;
    f<3>(s); // OK: Unqualified lookup now finds N2::f
                      //          then ADL kicks in because this name is unqualified
```

```
                    //          and finds N1::f
}
```

In the following contexts ADL-only lookup (that is, lookup in associated namespaces only) takes place:

- the lookup of non-member functions begin and end performed by the range-for loop if member lookup fails
- the dependent name lookup from the point of template instantiation.

- the lookup of non-member function get performed by structured binding declaration for tuple-like types (since C++17)

**Examples**

```
namespace A {
          struct X;
          struct Y;
          void f(int);
          void g(X);
}

namespace B {
       void f(int i) {
             f(i);      // calls B::f (endless recursion)
       }
       void g(A::X x) {
             g(x);      // Error: ambiguous between B::g (ordinary lookup)
                        //              and A::g (argument-dependent lookup)
       }
       void h(A::Y y) {
             h(y);      // calls B::h (endless recursion): ADL examines the A namespace
                        // but finds no A::h, so only B::h from ordinary lookup is used
       }
}
```

### 7.7 Overload resolution

In order to compile a function call, the compiler must first perform name lookup, which, for functions, may involve argument-dependent lookup, and for function templates may be followed by template argument deduction. If these steps produce more than one *candidate function*, then *overload resolution* is performed to select the function that will actually be called.

In general, the candidate function whose parameters match the arguments most closely is the one that is called.

For other contexts where overloaded function names can appear, see taking the address of an overloaded function.

1. 详细内容

Before overload resolution begins, the functions selected by name lookup and template argument deduction are combined to form the set of *candidate functions* (the exact criteria depend on the context in which overload resolution takes place, see below).

If any candidate function is a member function (static or non-static), but not a constructor, it is treated as if it has an extra parameter (*implicit object parameter*) which represents the object for which they are called and appears before the first of the actual parameters.

Similarly, the object on which a member function is being called is prepended to the argument list as the *implied object argument*.

For member functions of class X, the type of the implicit object parameter is affected by cv-qualifications and ref-qualifications of the member function as described in member functions.

The user-defined conversion functions are considered to be members of the *implied object argument* for the purpose of determining the type of the *implicit object parameter*.

The member functions introduced by a using-declaration into a derived class are considered to be members of the derived class for the purpose of defining the type of the *implicit* object parameter.

For the static member functions, the *implicit object parameter* is considered to match any object: its type is not examined and no conversion sequence is attempted for it.

For the rest of overload resolution, the *implied object argument* is indistinguishable from other arguments, but the following special rules apply to the *implicit object parameter*:

1) user-defined conversions cannot be applied to the implicit object parameter

2) rvalues can be bound to non-const implicit object parameter (unless this is for a ref-qualified member function)(since C++11) and do not affect the ranking of the implicit conversions.

```
struct B { void f(int); };

struct A { operator B&(); };

A a;

a.B::f(1); // Error: user-defined conversions cannot be applied

                    // to the implicit object parameter

static_cast<B&>(a).f(1); // OK
```

**Candidate functions**

The set of candidate functions and the list of arguments is prepared in a unique way for each of the contexts where overload resolution is used:

**Call to a named function**

If E in a function call expression E(args) names a set of overloaded functions and/or function templates (but not callable objects), the following rules are followed:

- If the expression E has the form PA->B or A.B (where A has class type cv T), then B is looked up as a member function of T. The function declarations found by that lookup are the candidate functions. The argument list for the purpose of overload resolution has the implied object argument of type cv T.

- If the expression E is a primary expression, the name is looked up following normal rules for function calls (which may involve ADL). The function declarations found by this lookup are (due to the way lookup works) either:

a) all non-member functions (in which case the argument list for the purpose of overload resolution is exactly the argument list used in the function call expression)

b) all member functions of some class T, in which case, if this is in scope and refers to T, *this is used as the implied object argument. Otherwise (if this is not in scope or does not point to T, a fake object of type T is used as the implied object argument, and if overload resolution subsequently selects a non-static member function, the program is ill-formed.

**Call to a class object**

If E in a function call expression E(args) has class type cv T, then

- The function-call operators of T are obtained by ordinary lookup of the name operator() in the context of the expression (E).operator(), and every declaration found is added to the set of candidate functions.
- For each non-explicit user-defined conversion function in T or in a base of T (unless hidden), whose cv-qualifiers is same or greater than T's cv-qualifiers, and where the conversion function converts to:
- to pointer-to-function
- to reference-to-pointer-to-function
- to reference-to-function

then a *surrogate call function* with a unique name whose first parameter is the result of the conversion, the remaining parameters are the parameter-list accepted by the result of the conversion, and the return type is the return type of the result of the conversion, is added to the set of candidate functions. If this surrogate function is selected by the subsequent overload resolution, then the user-defined conversion function will be called and then the result of the conversion will be called.

In any case, the argument list for the purpose of overload resolution is the argument list of the function call expression preceded by the implied object argument E (when matching against the surrogate function, the user-defined conversion will automatically convert the implied object argument to the first argument of the surrogate function).

```
int f1(int);

int f2(float);

struct A {

    using fp1 = int(*)(int);

    operator fp1() { return f1; } // conversion function to pointer to function

    using fp2 = int(*)(float);

    operator fp2() { return f2; } // conversion function to pointer to function

} a;

int i = a(1); // calls f1 via pointer returned from conversion function
```

**Call to an overloaded operator**

If at least one of the arguments to an operator in an expression has a class type or an enumeration type, both builtin operators and user-defined operator overloads participate in overload resolution, with the set of candidate functions selected as follows:

For a unary operator **@** whose argument has type T1 (after removing cv-qualifications), or binary operator **@** whose left operand has type T1 and right operand of type T2 (after removing cv-qualifications), three sets of candidate functions are prepared:

1) *member candidates*: if T1 is a complete class or a class currently being defined, the set of member candidates is the result of qualified name lookup of T1::operator@. In all other cases, the set of member candidates is empty.

2) *non-member candidates*: For the operators where operator overloading permits non-member forms, all declarations found by unqualified name lookup of operator@ in the context of the expression (which may involve ADL), except that member function declarations are ignored and do not prevent the lookup from continuing into the next enclosing scope. If both operands of a binary operator or the only operand of a unary operator has enumeration type, the only functions from the lookup set that become non-member candidates are the ones whose parameter has that enumeration type (or reference to that enumeration type)

3) *built-in candidates*: For operator,, the unary operator&, and the operator->, the set of built-in candidates is empty. For other operators built-in candidates are the ones listed in built-in operator pages as long as all operands can be implicitly converted to their parameters. If any built-in candidate has the same parameter list as a non-member candidate that isn't a function template specialization, it is not added to the list of built-in candidates. When the built-in assignment operators are considered, the conversions from their left-hand arguments are restricted: user-defined conversions are not considered.

The set of candidate functions to be submitted for overload resolution is a union of the three sets above. The argument list for the purpose of overload resolution consists of the operands of the operator except for operator->, where the second operand is not an argument for the function call (see member access operator).

```
struct A {
    operator int(); // user-defined conversion
};
A operator+(const A&, const A&); // non-member user-defined operator
void m()
{
    A a, b;
    a + b; // member-candidates: none
           // non-member candidates: operator+(a,b)
           // built-in candidates: int(a) + int(b)
           // overload resolution chooses operator+(a,b)
}
```

If the overload resolution selects a built-in candidate, the user-defined conversion sequence from an operand of class type is not allowed to have a second standard conversion sequence: the user-defined conversion function must give the expected operand type directly:

```
struct Y { operator int*(); };    // Y is convertible to int*
int *a = Y() + 100.0; // error: no operator+ between pointer and double
```

For operator,, the unary operator&, and operator->, if there are no viable functions (see below) in the set of candidate functions, then the operator is reinterpreted as a built-in.

**Initialization by constructor**

When an object of class type is direct-initialized or default-initialized outside a copy-initialization context, the candidate functions are all constructors of the class being initialized. The argument list is the expression list of the initializer.

When an object of class type is copy-initialized from an object of the same or derived class type, or default-initialized in a copy-initialization context, the candidate functions are all converting constructors of the class being initialized. The argument list is the expression of the initializer.

**Copy-initialization by conversion**

If copy-initialization of an object of class type requires that a user-defined conversion function is called to convert the initializer expression of type cv S to the type cv T of the object being initialized, the following functions are candidate functions:

- all converting constructors of T
- the non-explicit conversion functions from S and its base classes (unless hidden) to T or class derived from T or a reference to such. If this copy-initialization is part of the direct-initialization sequence of *cv* T (initializing a reference to be bound to the first parameter of a constructor that takes a reference to cv T), then explicit conversion functions are also considered.

Either way, the argument list for the purpose of overload resolution consists of a single argument which is the initializer expression, which will be compared against the first argument of the constructor or against the implicit object argument of the conversion function.

**Non-class initialization by conversion**

When initialization of an object of non-class type cv1 T requires a user-defined conversion function to convert from an initializer expression of class type cv S, the following functions are candidates:

- the non-explicit user-defined conversion functions of S and its base classes (unless hidden) that produce type T or a type convertible to T by a standard conversion sequence, or a reference to such type. cv qualifiers on the returned type are ignored for the purpose of selecting candidate functions.
- if this is direct-initialization, the explicit user-defined conversion functions of S and its base classes (unless hidden) that produce type T or a type convertible to T by a qualification conversion, or a reference to such type, are also considered.

Either way, the argument list for the purpose of overload resolution consists of a single argument which is the initializer expression, which will be compared against the implicit object argument of the conversion function.

**Reference initialization by conversion**

During reference initialization, where the reference to cv1 T is bound to the glvalue or class prvalue result of a conversion from the initializer expression from the class type cv2 S, the following functions are selected for the candidate set:

- the non-explicit user-defined conversion functions of S and its base classes (unless hidden) to the type
- (when initializing lvalue reference or rvalue reference to function) lvalue reference to cv2 T2
- (when initializing rvalue reference or lvalue reference to function) cv2 T2 or rvalue reference to cv2 T2

where cv2 T2 is reference-compatible with cv1 T

- for direct initializaton, the explicit user-defined conversion functions are also considered if T2 is the same type as T or can be converted to type T with a qualification conversion.

Either way, the argument list for the purpose of overload resolution consists of a single argument which is the initializer expression, which will be compared against the implicit object argument of the conversion function.

**List-initialization**

When an object of non-aggregate class type T is list-initialized, two-phase overload resolution takes place.

- at phase 1, the candidate functions are all initializer-list constructors of T and the argument list for the purpose of overload resolution consists of a single initializer list argument
- if overload resolution fails at phase 1, phase 2 is entered, where the candidate functions are all constructors of Tand the argument list for the purpose of overload resolution consists of the individual elements of the initializer list.

If the initializer list is empty and T has a default constructor, phase 1 is skipped.

In copy-list-initialization, if phase 2 selects an explicit constructor, the initialization is ill-formed (as opposed to all over copy-initializations where explicit constructors are not even considered).

**Viable functions**

Given the set of candidate functions, constructed as described above, the next step of overload resolution is examining arguments and parameters to reduce the set to the set of *viable functions*

To be included in the set of viable functions, the candidate function must satisfy the following:

1) If there are M arguments, the candidate function that has exactly M parameters is viable

2) If the candidate function has less than M parameters, but has an ellipsis parameter, it is viable.

3) If the candidate function has more than M parameters and the M+1'st parameter and all parameters that follow must have default arguments, it is viable. For the rest of overload resolution, the parameter list is truncated at M.

| | |
|---|---|
| 4) If the function has an associated constraint, it must be satisfied | (since C++20) |

5) For every argument there must be at least one implicit conversion sequence that converts it to the corresponding parameter.

6) If any parameter has reference type, reference binding is accounted for at this step: if an rvalue argument corresponds to non-const lvalue reference parameter or an lvalue argument corresponds to rvalue reference parameter, the function is not viable.

User-defined conversions (both converting constructors and user-defined conversion functions) are prohibited from taking part in implicit conversion sequence where it would make it possible to apply more than one user-defined conversion. Specifically, they are prohibited if the target of the conversion is the first parameter of a constructor or the implicit object parameter of a user-defined conversion function, and that constructor/user-defined conversion is a candidate for

- copy-initialization of a class by user-defined conversion,
- initialization by a conversion function,
- initialization by conversion function for direct reference binding,
- initialization by constructor where the argument is a temporary in class copy-initialization,
- initialization by list-initialization where the initializer list has exactly one element that is itself an initializer list, and the target is the first parameter of a constructor of class X, and the conversion is to X or reference to (possibly cv-qualified) X

```
struct A { A(int); };

struct B { B(A); };

B b{ {0} }; // list-init of B

// candidates: B(const B&), B(B&&), B(A)

// {0} -> B&& not viable: would have to call B(A)

// {0} -> const B&: not viable: would have to bind to rvalue, would have to call B(A)

// {0} -> A viable. Calls A(int): user-defined conversion to A is not banned
```

**Best viable function**

For each pair of viable function F1 and F2, the implicit conversion sequences from the i-th argument to i-th parameter are ranked to determine which one is better (except the first argument, the *implicit object argument* for static member functions has no effect on the ranking)

F1 is determined to be a better function than F2 if implicit conversions for all arguments of F1 are *not worse* than the implicit conversions for all arguments of F2, and

1) there is at least one argument of F1 whose implicit conversion is *better* than the corresponding implicit conversion for that argument of F2

2) or. if not that, (only in context of non-class initialization by conversion), the standard conversion sequence from the return type of F1 to the type being initialized is *better* than the standard conversion sequence from the return type of F2

3) or, if not that, F1 is a non-template function while F2 is a template specialization

4) or, if not that, F1 and F2 are both template specializations and F1 is *more specialized* according to the partial ordering rules for template specializations

5) or, if not that, F1 and F2 are non-template functions with the same parameter-type-lists, and F1 is more constrained than F2 according to the partial ordering of constraints (since C++20)

6) or, if not that, F1 is generated from a user-defined deduction-guide and F2 is not

7) or, if not that, F1 is the copy deduction candidate and F2 is not

8) or, if not that, F1 is generated from a non-template constructor and F2 is generated from a constructor template

```
template <class T> struct A {
    using value_type = T;
    A(value_type);                              // #1
    A(const A&);                                // #2
    A(T, T, int);                               // #3
    template<class U> A(int, T, U); // #4
};                                              // #5 is A(A), the copy deduction candidate
A x (1, 2, 3);    // uses #3, generated from a non-template constructor
template <class T> A(T) -> A<T>;    // #6, less specialized than #5
A a (42); // uses #6 to deduce A<int> and #1 to initialize
A b = a;    // uses #5 to deduce A<int> and #2 to initialize
template <class T> A(A<T>) -> A<A<T>>;    // #7, as specialized as #5
A b2 = a;    // uses #7 to deduce A<A<int>> and #1 to initialize
```

These pair-wise comparisons are applied to all viable functions. If exactly one viable function is better than all others, overload resolution succeeds and this function is called. Otherwise, compilation fails.

```
void Fcn(const int*, short); // overload #1

void Fcn(int*, int); // overload #2

int i;

short s = 0;

void f() {

    Fcn(&i, 1L);    // 1st argument: &i -> int* is better than &i -> const int*

                    // 2nd argument: 1L -> short and 1L -> int are equivalent

                    // calls Fcn(int*, int)


    Fcn(&i,'c');    // 1st argument: &i -> int* is better than &i -> const int*

                    // 2nd argument: 'c' -> int is better than 'c' -> short

                    // calls Fcn(int*, int)


    Fcn(&i, s);     // 1st argument: &i -> int* is better than &i -> const int*

                    // 2nd argument: s -> short is better than s -> int

                    // no winner, compilation error

}
```

**Ranking of implicit conversion sequences**

The argument-parameter implicit conversion sequences considered by overload resolution correspond to implicit conversions used in copy initialization (for non-reference parameters), except that when considering conversion to the implicit object parameter or to the left-hand side of assignment operator, conversions that create temporary objects are not considered.

Each type of standard conversion sequence is assigned one of three ranks:

1) **Exact match**: no conversion required, lvalue-to-rvalue conversion, qualification conversion, function pointer conversion, (since C++17) user-defined conversion of class type to the same class

2) **Promotion**: integral promotion, floating-point promotion

3) **Conversion**: integral conversion, floating-point conversion, floating-integral conversion, pointer conversion, pointer-to-member conversion, boolean conversion, user-defined conversion of a derived class to its base

The rank of the standard conversion sequence is the worst of the ranks of the standard conversions it holds (there may be up to three conversions)

Binding of a reference parameter directly to the argument expression is either Identity or a derived-to-base Conversion:

```
struct Base {};

struct Derived : Base {} d;

int f(Base&);       // overload #1

int f(Derived&); // overload #2
int i = f(d); // d -> Derived& has rank Exact Match
              // d -> Base& has rank Conversion
              // calls f(Derived&)
```

Since ranking of conversion sequences operates with types and value categories only, a bit field can bind to a reference argument for the purpose of ranking, but if that function gets selected, it will be ill-formed.

1) A standard conversion sequence is always *better* than a user-defined conversion sequence or an ellipsis conversion sequence.

2) A user-defined conversion sequence is always *better* than an ellipsis conversion sequence

3) A standard conversion sequence S1 is *better* than a standard conversion sequence S2 if

a) S1 is a subsequence of S2, excluding lvalue transformations. The identity conversion sequence is considered a subsequence of any other conversion

b) Or, if not that, the rank of S1 is better than the rank of S2

c) or, if not that, both S1 and S2 are binding to a reference parameter to something other than the implicit object parameter of a ref-qualified member function, and S1 binds an rvalue reference to an rvalue while S2binds an lvalue reference to an rvalue

```
int i;
int f1();
int g(const int&);   // overload #1
int g(const int&&); // overload #2
int j = g(i);       // lvalue int -> const int& is the only valid conversion
int k = g(f1()); // rvalue int -> const int&& better than rvalue int -> const int&
```

d) or, if not that, both S1 and S2 are binding to a reference parameter and S1 binds an lvalue reference to function while S2 binds an rvalue reference to function.

```
int f(void(&)());   // overload #1
int f(void(&&)()); // overload #2
void g();
int i1 = f(g);       // calls #1
```

e) or, if not that, both S1 and S2 are binding to a reference parameters only different in top-level cv-qualification, and S1's type is less cv-qualified than S2's.

```
int f(const int &); // overload #1
int f(int &);          // overload #2 (both references)
int g(const int &); // overload #1
int g(int);             // overload #2
int i;
int j = f(i); // lvalue i -> int& is better than lvalue int -> const int&

                     // calls f(int&)

int k = g(i); // lvalue i -> const int& ranks Exact Match

                     // lvalue i -> rvalue int ranks Exact Match

                     // ambiguous overload: compilation error
```

f) Or, if not that, S1 and S2 only differ in qualification conversion, and the cv-qualification of the result of S1 is a subset of the cv-qualification of the result of S2

```
int f(const int*);

int f(int*);

int i;

int j = f(&i); // &i -> int* is better than &i -> const int*, calls f(int*)
```

4) A user-defined conversion sequence U1 is better than a user-defined conversion sequence U2 if they call the same constructor/user-defined conversion function or initialize the same class with aggregate-initialization, and in either case the second standard conversion sequence in U1 is better than the second standard conversion sequence in U2

```
struct A {

      operator short(); // user-defined conversion function

} a;

int f(int);      // overload #1

int f(float); // overload #2

int i = f(a); // A -> short, followed by short -> int (rank Promotion)

                     // A -> short, followed by short -> float (rank Conversion)

                     // calls f(int)
```

5) A list-initialization sequence   L1 is *better* than list-initialization sequence L2 if L1 initializes an std::initializer_list parameter, while L2 does not.

```
void f1(int);                                                    // #1

void f1(std::initializer_list<long>);                 // #2

void g1() { f1({42}); }                                     // chooses #2


void f2(std::pair<const char*, const char*>); // #3

void f2(std::initializer_list<std::string>);    // #4

void g2() { f2({"foo","bar"}); }                      // chooses #4
```

6) A list-initialization sequence L1 is better than list-initialization sequence L2 if the corresponding parameters are references to arrays, and L1 converts to type "array of N1 T," L2 converts to type "array of N2 T", and N1 is smaller than N2.

7.8 Operator overloading

7.9 重载集的地址

Besides function-call expressions, where overload resolution takes place, the name of an overloaded function may appear in the following 7 contexts:

除了函数调用表达式（其中发生重载解析）之外，重载函数的名称可能会出现在以下 7 个上下文中：

1) 对象或引用的初始化中

2) 赋值表达式的右边

3) 作为函数调用参数

4) 作为用户自定义操作符参数

5) return 语句

6) explicit cast 或 static cast 参数

7) 无类型 template argument

In each context, the name of an overloaded function may be preceded by address-of operator **&** and may be enclosed in a redundant set of parentheses.

In all these contexts, the function selected from the overload set is the function whose type matches the pointer to function, reference to function, or pointer to member function type that is expected by *target*: the object or reference being initialized, the left-hand side of the assignment, function or operator parameter, the return type of a function, the target type of a cast, or the type of the template parameter, respectively.

The type of the function must match the target exactly, no implicit conversions are considered (e.g. a function returning a pointer to derived won't get selected when initializing a pointer to function returning a pointer to base)

If the function name names a function template, then, first, template argument deduction is done, and if it succeeds, it produces a single template specialization which is added to the set of overloads to consider. All functions whose associated constraints are not satisfied are dropped from the set. (since C++20) If more than one function from the set matches the target, and at least one function is non-template, the template specializations are eliminated from consideration. For any pair of non-template functions where one is more constrained than another, the less constrained function is dropped from the set (since C++20). If all remaining candidates are template specializations, less specialized ones are removed if more specialized are available. If more than one candidate remains after the removals, the program is ill-formed.

## 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.