Оригинал:

вырезка из стандарта C++

N4659 “Working Draft, Standard for Programming Language C++”

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf

**Program execution**

1 The semantic descriptions in this International Standard define a parameterized nondeterministic abstract machine. This International Standard places no requirement on the structure of conforming implementations. In particular, they need not copy or emulate the structure of the abstract machine. Rather, conforming implementations are required to emulate (only) the observable behavior of the abstract machine as explained below.6

2 Certain aspects and operations of the abstract machine are described in this International Standard as implementation-defined (for example, sizeof(int)). These constitute the parameters of the abstract machine. Each implementation shall include documentation describing its characteristics and behavior in these respects. Such documentation shall define the instance of the abstract machine that corresponds to that implementation (referred to as the “corresponding instance” below).

3 Certain other aspects and operations of the abstract machine are described in this International Standard as unspecified (for example, evaluation of expressions in a new-initializer if the allocation function fails to allocate memory (8.3.4)). Where possible, this International Standard defines a set of allowable behaviors. These define the nondeterministic aspects of the abstract machine. An instance of the abstract machine can thus have more than one possible execution for a given program and a given input.

4 Certain other operations are described in this International Standard as undefined (for example, the effect of attempting to modify a const object). [Note: This International Standard imposes no requirements on the behavior of programs that contain undefined behavior. — end note]

5 A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible executions of the corresponding instance of the abstract machine with the same program and the same input. However, if any such execution contains an undefined operation, this International Standard places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation).

6 An instance of each object with automatic storage duration (6.7.3) is associated with each entry into its block. Such an object exists and retains its last-stored value during the execution of the block and while the block is suspended (by a call of a function or receipt of a signal).

7 The least requirements on a conforming implementation are:

(7.1) — Accesses through volatile glvalues are evaluated strictly according to the rules of the abstract machine.

(7.2) — At program termination, all data written into files shall be identical to one of the possible results that execution of the program according to the abstract semantics would have produced.

(7.3) — The input and output dynamics of interactive devices shall take place in such a fashion that prompting output is actually delivered before a program waits for input. What constitutes an interactive device is implementation-defined. These collectively are referred to as the observable behavior of the program. [Note: More stringent correspondences between abstract and actual semantics may be defined by each implementation. — end note]

8 [Note: Operators can be regrouped according to the usual mathematical rules only where the operators really are associative or commutative. For example, in the following fragment

int a, b;

/\* ... \*/

a = a + 32760 + b + 5;

the expression statement behaves exactly the same as

a = (((a + 32760) + b) + 5);

due to the associativity and precedence of these operators. Thus, the result of the sum (a + 32760) is next added to b, and that result is then added to 5 which results in the value assigned to a. On a machine in which overflows produce an exception and in which the range of values representable by an int is [-32768, +32767], the implementation cannot rewrite this expression as

a = ((a + b) + 32765);

since if the values for a and b were, respectively, -32754 and -15, the sum a + b would produce an exception while the original expression would not; nor can the expression be rewritten either as

a = ((a + 32765) + b);

or

a = (a + (b + 32765));

since the values for a and b might have been, respectively, 4 and -8 or -17 and 12. However on a machine in which overflows do not produce an exception and in which the results of overflows are reversible, the above expression statement can be rewritten by the implementation in any of the above ways because the same result will occur. — end note]

9 A constituent expression is defined as follows:

(9.1) — The constituent expression of an expression is that expression.

(9.2) — The constituent expressions of a braced-init-list or of a (possibly parenthesized) expression-list are the constituent expressions of the elements of the respective list.

(9.3) — The constituent expressions of a brace-or-equal-initializer of the form = initializer-clause are the constituent expressions of the initializer-clause.

[Example:

struct A { int x; };

struct B { int y; struct A a; };

B b = { 5, { 1+1 } };

The constituent expressions of the initializer used for the initialization of b are 5 and 1+1.

— end example]

10 The immediate subexpressions of an expression e are

(10.1) — the constituent expressions of e’s operands (Clause 8),

(10.2) — any function call that e implicitly invokes,

(10.3) — if e is a lambda-expression (8.1.5), the initialization of the entities captured by copy and the constituent expressions of the initializer of the init-captures,

(10.4) — if e is a function call (8.2.2) or implicitly invokes a function, the constituent expressions of each default argument (11.3.6) used in the call, or

(10.5) — if e creates an aggregate object (11.6.1), the constituent expressions of each default member initializer (12.2) used in the initialization.

11 A subexpression of an expression e is an immediate subexpression of e or a subexpression of an immediate subexpression of e. [Note: Expressions appearing in the compound-statement of a lambda-expression are not subexpressions of the lambda-expression. — end note]

12 A full-expression is

(12.1) — an unevaluated operand (Clause 8),

(12.2) — a constant-expression (8.20),

(12.3) — an init-declarator (Clause 11) or a mem-initializer (15.6.2), including the constituent expressions of the initializer,

(12.4) — an invocation of a destructor generated at the end of the lifetime of an object other than a temporary object (15.2), or

(12.5) — an expression that is not a subexpression of another expression and that is not otherwise part of a full-expression. If a language construct is defined to produce an implicit call of a function, a use of the language construct is considered to be an expression for the purposes of this definition. Conversions applied to the result of an expression in order to satisfy the requirements of the language construct in which the expression appears are also considered to be part of the full-expression. For an initializer, performing the initialization of the entity (including evaluating default member initializers of an aggregate) is also considered part of the full-expression.

[Example:

struct S {

S(int i): I(i) { } // full-expression is initialization of I

int& v() { return I; }

~S() noexcept(false) { }

private:

int I;

};

S s1(1); // full-expression is call of S::S(int)

void f() {

S s2 = 2; // full-expression is call of S::S(int)

if (S(3).v()) // full-expression includes lvalue-to-rvalue and

// int to bool conversions, performed before

// temporary is deleted at end of full-expression

{ }

bool b = noexcept(S()); // exception specification of destructor of S

// considered for noexcept

// full-expression is destruction of s2 at end of block

}

struct B {

B(S = S(0));

};

B b[2] = { B(), B() }; // full-expression is the entire initialization

// including the destruction of temporaries

— end example]

13 [Note: The evaluation of a full-expression can include the evaluation of subexpressions that are not lexically part of the full-expression. For example, subexpressions involved in evaluating default arguments (11.3.6) are considered to be created in the expression that calls the function, not the expression that defines the default argument. — end note]

14 Reading an object designated by a volatile glvalue (6.10), modifying an object, calling a library I/O function, or calling a function that does any of those operations are all side effects, which are changes in the state of the execution environment. Evaluation of an expression (or a subexpression) in general includes both value computations (including determining the identity of an object for glvalue evaluation and fetching a value previously assigned to an object for prvalue evaluation) and initiation of side effects. When a call to a library I/O function returns or an access through a volatile glvalue is evaluated the side effect is considered complete, even though some external actions implied by the call (such as the I/O itself) or by the volatile access may not have completed yet.

15 Sequenced before is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread (4.7), which induces a partial order among those evaluations. Given any two evaluations A and B, if A is sequenced before B (or, equivalently, B is sequenced after A), then the execution of A shall precede the execution of B. If A is not sequenced before B and B is not sequenced before A, then A and B are unsequenced. [Note: The execution of unsequenced evaluations can overlap. — end note] Evaluations A and B are indeterminately sequenced when either A is sequenced before B or B is sequenced before A, but it is unspecified which. [Note: Indeterminately sequenced evaluations cannot overlap, but either could be executed first. — end note] An expression X is said to be sequenced before an expression Y if every value computation and every side effect associated with the expression X is sequenced before every value computation and every side effect associated with the expression Y.

16 Every value computation and side effect associated with a full-expression is sequenced before every value computation and side effect associated with the next full-expression to be evaluated.

17 Except where noted, evaluations of operands of individual operators and of subexpressions of individual expressions are unsequenced. [Note: In an expression that is evaluated more than once during the execution of a program, unsequenced and indeterminately sequenced evaluations of its subexpressions need not be performed consistently in different evaluations. — end note] The value computations of the operands of an operator are sequenced before the value computation of the result of the operator. If a side effect on a memory location (4.4) is unsequenced relative to either another side effect on the same memory location or a value computation using the value of any object in the same memory location, and they are not potentially concurrent (4.7), the behavior is undefined. [Note: The next section imposes similar, but more complex restrictions on potentially concurrent computations. — end note]

[Example:

void g(int i) {

i = 7, i++, i++; // i becomes 9

i = i++ + 1; // the value of i is incremented

i = i++ + i; // the behavior is undefined

i = i + 1; // the value of i is incremented

}

— end example]

18 When calling a function (whether or not the function is inline), every value computation and side effect associated with any argument expression, or with the postfix expression designating the called function, is sequenced before execution of every expression or statement in the body of the called function. For each function invocation F, for every evaluation A that occurs within F and every evaluation B that does not occur within F but is evaluated on the same thread and as part of the same signal handler (if any), either A is sequenced before B or B is sequenced before A. [Note: If A and B would not otherwise be sequenced then they are indeterminately sequenced. — end note] Several contexts in C++ cause evaluation of a function call, even though no corresponding function call syntax appears in the translation unit. [ Example: Evaluation of a new-expression invokes one or more allocation and constructor functions; see 8.3.4. For another example, invocation of a conversion function (15.3.2) can arise in contexts in which no function call syntax appears. — end example] The sequencing constraints on the execution of the called function (as described above) are features of the function calls as evaluated, whatever the syntax of the expression that calls the function might be.

19 If a signal handler is executed as a result of a call to the std::raise function, then the execution of the handler is sequenced after the invocation of the std::raise function and before its return. [Note: When a signal is received for another reason, the execution of the signal handler is usually unsequenced with respect to the rest of the program. — end note]

**Object lifetime**

1 The lifetime of an object or reference is a runtime property of the object or reference. An object is said to have non-vacuous initialization if it is of a class or aggregate type and it or one of its subobjects is initialized by a constructor other than a trivial default constructor. [Note: Initialization by a trivial copy/move constructor is non-vacuous initialization. — end note] The lifetime of an object of type T begins when:

(1.1) — storage with the proper alignment and size for type T is obtained, and

(1.2) — if the object has non-vacuous initialization, its initialization is complete,

except that if the object is a union member or subobject thereof, its lifetime only begins if that union member is the initialized member in the union (11.6.1, 15.6.2), or as described in 12.3. The lifetime of an object o of type T ends when:

(1.3) — if T is a class type with a non-trivial destructor (15.4), the destructor call starts, or

(1.4) — the storage which the object occupies is released, or is reused by an object that is not nested within o (4.5).

2 The lifetime of a reference begins when its initialization is complete. The lifetime of a reference ends as if it were a scalar object.

3 [Note: 15.6.2 describes the lifetime of base and member subobjects. — end note]

4 The properties ascribed to objects and references throughout this International Standard apply for a given object or reference only during its lifetime. [Note: In particular, before the lifetime of an object starts and after its lifetime ends there are significant restrictions on the use of the object, as described below, in 15.6.2 and in 15.7. Also, the behavior of an object under construction and destruction might not be the same as the behavior of an object whose lifetime has started and not ended. 15.6.2 and 15.7 describe the behavior of objects during the construction and destruction phases. — end note]

5 A program may end the lifetime of any object by reusing the storage which the object occupies or by explicitly calling the destructor for an object of a class type with a non-trivial destructor. For an object of a class type with a non-trivial destructor, the program is not required to call the destructor explicitly before the storage which the object occupies is reused or released; however, if there is no explicit call to the destructor or if a delete-expression (8.3.5) is not used to release the storage, the destructor shall not be implicitly called and any program that depends on the side effects produced by the destructor has undefined behavior.

6 Before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any pointer that represents the address of the storage location where the object will be or was located may be used but only in limited ways. For an object under construction or destruction, see 15.7. Otherwise, such a pointer refers to allocated storage (6.7.4.2), and using the pointer as if the pointer were of type void\*, is well-defined. Indirection through such a pointer is permitted but the resulting lvalue may only be used in limited ways, as described below. The program has undefined behavior if:

(6.1) — the object will be or was of a class type with a non-trivial destructor and the pointer is used as the operand of a delete-expression,

(6.2) — the pointer is used to access a non-static data member or call a non-static member function of the object, or

(6.3) — the pointer is implicitly converted (7.11) to a pointer to a virtual base class, or

(6.4) — the pointer is used as the operand of a static\_cast (8.2.9), except when the conversion is to pointer to cv void, or to pointer to cv void and subsequently to pointer to cv char, cv unsigned char, or cv std::byte (21.2.1), or

(6.5) — the pointer is used as the operand of a dynamic\_cast (8.2.7).

[Example:

#include <cstdlib>

struct B {

virtual void f();

void mutate();

virtual ~B();

};

struct D1 : B { void f(); };

struct D2 : B { void f(); };

void B::mutate() {

new (this) D2; // reuses storage — ends the lifetime of \*this

f(); // undefined behavior

... = this; // OK, this points to valid memory

}

void g() {

void\* p = std::malloc(sizeof(D1) + sizeof(D2));

B\* pb = new (p) D1;

pb->mutate();

\*pb; // OK: pb points to valid memory

void\* q = pb; // OK: pb points to valid memory

pb->f(); // undefined behavior, lifetime of \*pb has ended

}

— end example]

7 Similarly, before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any glvalue that refers to the original object may be used but only in limited ways. For an object under construction or destruction, see 15.7. Otherwise, such a glvalue refers to allocated storage (6.7.4.2), and using the properties of the glvalue that do not depend on its value is well-defined. The program has undefined behavior if:

(7.1) — the glvalue is used to access the object, or

(7.2) — the glvalue is used to call a non-static member function of the object, or

(7.3) — the glvalue is bound to a reference to a virtual base class (11.6.3), or

(7.4) — the glvalue is used as the operand of a dynamic\_cast (8.2.7) or as the operand of typeid.

8 If, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, a new object is created at the storage location which the original object occupied, a pointer that pointed to the original object, a reference that referred to the original object, or the name of the original object will automatically refer to the new object and, once the lifetime of the new object has started, can be used to manipulate the new object, if:

(8.1) — the storage for the new object exactly overlays the storage location which the original object occupied, and

(8.2) — the new object is of the same type as the original object (ignoring the top-level cv-qualifiers), and

(8.3) — the type of the original object is not const-qualified, and, if a class type, does not contain any non-static data member whose type is const-qualified or a reference type, and

(8.4) — the original object was a most derived object (4.5) of type T and the new object is a most derived object of type T (that is, they are not base class subobjects).

[Example:

struct C {

int i;

void f();

const C& operator=( const C& );

};

const C& C::operator=( const C& other) {

if ( this != &other ) {

this->~C(); // lifetime of \*this ends

new (this) C(other); // new object of type C created

f(); // well-defined

}

return \*this;

}

C c1;

C c2;

c1 = c2; // well-defined

c1.f(); // well-defined; c1 refers to a new object of type C

— end example] [Note: If these conditions are not met, a pointer to the new object can be obtained from a pointer that represents the address of its storage by calling std::launder (21.6). — end note]

9 If a program ends the lifetime of an object of type T with static (6.7.1), thread (6.7.2), or automatic (6.7.3) storage duration and if T has a non-trivial destructor, the program must ensure that an object of the original type occupies that same storage location when the implicit destructor call takes place; otherwise the behavior of the program is undefined. This is true even if the block is exited with an exception.

[Example:

class T { };

struct B {

~B();

};

void h() {

B b;

new (&b) T;

} // undefined behavior at block exit

— end example]

10 Creating a new object within the storage that a const complete object with static, thread, or automatic storage duration occupies, or within the storage that such a const object used to occupy before its lifetime ended, results in undefined behavior.

[Example:

struct B {

B();

~B();

};

const B b;

void h() {

b.~B();

new (const\_cast(&b)) const B; // undefined behavior

}

— end example]

11 In this section, “before” and “after” refer to the “happens before” relation (4.7). [Note: Therefore, undefined behavior results if an object that is being constructed in one thread is referenced from another thread without adequate synchronization. — end note]

**Types**

1 [Note: 6.9 and the subclauses thereof impose requirements on implementations regarding the representation of types. There are two kinds of types: fundamental types and compound types. Types describe objects (4.5), references (11.3.2), or functions (11.3.5). — end note]

2 For any object (other than a base-class subobject) of trivially copyable type T, whether or not the object holds a valid value of type T, the underlying bytes (4.4) making up the object can be copied into an array of char, unsigned char, or std::byte (21.2.1). If the content of that array is copied back into the object, the object shall subsequently hold its original value.

[Example:

#define N sizeof(T)

char buf[N];

T obj; // obj initialized to its original value

std::memcpy(buf, &obj, N); // between these two calls to std::memcpy, obj might be modified

std::memcpy(&obj, buf, N); // at this point, each subobject of obj of scalar type holds its original value

— end example]

3 For any trivially copyable type T, if two pointers to T point to distinct T objects obj1 and obj2, where neither obj1 nor obj2 is a base-class subobject, if the underlying bytes (4.4) making up obj1 are copied into obj2, obj2 shall subsequently hold the same value as obj1.

[Example:

T\* t1p;

T\* t2p;

// provided that t2p points to an initialized object ...

std::memcpy(t1p, t2p, sizeof(T));

// at this point, every subobject of trivially copyable type in \*t1p contains

// the same value as the corresponding subobject in \*t2p

— end example]

4 The object representation of an object of type T is the sequence of N unsigned char objects taken up by the object of type T, where N equals sizeof(T). The value representation of an object is the set of bits that hold the value of type T. For trivially copyable types, the value representation is a set of bits in the object representation that determines a value, which is one discrete element of an implementation-defined set of values.

5 A class that has been declared but not defined, an enumeration type in certain contexts (10.2), or an array of unknown bound or of incomplete element type, is an incompletely-defined object type. 46 Incompletely-defined object types and cv void are incomplete types (6.9.1). Objects shall not be defined to have an incomplete type.

6 A class type (such as “class X”) might be incomplete at one point in a translation unit and complete later on; the type “class X” is the same type at both points. The declared type of an array object might be an array of incomplete class type and therefore incomplete; if the class type is completed later on in the translation unit, the array type becomes complete; the array type at those two points is the same type. The declared type of an array object might be an array of unknown bound and therefore be incomplete at one point in a translation unit and complete later on; the array types at those two points (“array of unknown bound of T” and “array of N T”) are different types. The type of a pointer to array of unknown bound, or of a type defined by a typedef declaration to be an array of unknown bound, cannot be completed.

[Example:

class X; // X is an incomplete type

extern X\* xp; // xp is a pointer to an incomplete type

extern int arr[]; // the type of arr is incomplete

typedef int UNKA[]; // UNKA is an incomplete type

UNKA\* arrp; // arrp is a pointer to an incomplete type

UNKA\*\* arrpp;

void foo() {

xp++; // ill-formed: X is incomplete

arrp++; // ill-formed: incomplete type

arrpp++; // OK: sizeof UNKA\* is known

}

struct X { int i; }; // now X is a complete type

int arr[10]; // now the type of arr is complete

X x;

void bar() {

xp = &x; // OK; type is “pointer to X”

arrp = &arr; // ill-formed: different types

xp++; // OK: X is complete

++; // ill-formed: UNKA can’t be completed

}

— end example]

7 [Note: The rules for declarations and expressions describe in which contexts incomplete types are prohibited. — end note]

8 An object type is a (possibly cv-qualified) type that is not a function type, not a reference type, and not cv void.

9 Arithmetic types (6.9.1), enumeration types, pointer types, pointer to member types (6.9.2), std::nullptr\_- t, and cv-qualified (6.9.3) versions of these types are collectively called scalar types. Scalar types, POD classes (Clause 12), arrays of such types and cv-qualified versions of these types are collectively called POD types. Cv-unqualified scalar types, trivially copyable class types (Clause 12), arrays of such types, and cv-qualified versions of these types are collectively called trivially copyable types. Scalar types, trivial class types (Clause 12), arrays of such types and cv-qualified versions of these types are collectively called trivial types. Scalar types, standard-layout class types (Clause 12), arrays of such types and cv-qualified versions of these types are collectively called standard-layout types.

10 A type is a literal type if it is:

(10.1) — possibly cv-qualified void; or

(10.2) — a scalar type; or

(10.3) — a reference type; or

(10.4) — an array of literal type; or

(10.5) — a possibly cv-qualified class type (Clause 12) that has all of the following properties:

(10.5.1) — it has a trivial destructor,

(10.5.2) — it is either a closure type (8.1.5.1), an aggregate type (11.6.1), or has at least one constexpr constructor or constructor template (possibly inherited (10.3.3) from a base class) that is not a copy or move constructor,

(10.5.3) — if it is a union, at least one of its non-static data members is of non-volatile literal type, and

(10.5.4) — if it is not a union, all of its non-static data members and base classes are of non-volatile literal types. [Note: A literal type is one for which it might be possible to create an object within a constant expression. It is not a guarantee that it is possible to create such an object, nor is it a guarantee that any object of that type will usable in a constant expression. — end note]

11 Two types cv1 T1 and cv2 T2 are layout-compatible types if T1 and T2 are the same type, layout-compatible enumerations (10.2), or layout-compatible standard-layout class types (12.2).