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]

Перевод:

**Выполнение программы**

1 Семантическое описание в данном Международном стандарте определяет параметризованную недетерминированную абстрактную машину. Данный Международный стандарт не накладывает ограничений к структуре соответствующих реализаций. В частности, они не обязаны копировать или эмулировать структуру абстрактной машины. Скорее всего, соответствующим реализациям эмуляция потребуется (только) для наблюдения за поведением абстрактной машины, как описано ниже в пункте 6.

2 Ряд частей и операций абстрактной машины описаны в данном Международном стандарте как определенные реализацией (например, sizeof(int)). Они составляют параметры абстрактной машины. Каждая реализация должна включать документацию, которая описывает характеристики и поведение в таких случаях. Такая документация должна определять экземпляр абстрактной машины, который соответствует данной реализации (именуемый ниже, как “соответствующий экземпляр”).

3 Ряд других частей и операций абстрактной машины описаны в данном Международном стандарте как неопределенные (например, расчет выражений в инициализаторе new, если функция выделения памяти не может ее выделить (8.3.4)). Там, где это возможно, данный Международный стандарт определяет набор допустимых моделей поведения. Это определяет недетерминированные части абстрактной машины. Таким образом, экземпляр абстрактной машины может иметь больше одного возможного варианта исполнения для заданных программы и входных параметров.

4 Ряд других операций описаны в данном Международном стандарте как неопределенные (например, результат модификации константного объекта). [Прим.: Данный международный стандарт не предъявляет требований к поведению программ, которые содержат неопределенное поведение. – конец прим.]

5 Соответствующая реализация, выполняя хорошо сконструированную программу, должна обеспечивать такое наблюдаемое поведение, которое наблюдается при одном из возможных вызовов соответствующего экземпляра абстрактной машины с той же программой и теми же входными параметрами. Однако, если какой-либо из таких вызовов содержит неопределенное поведение, данный Международный стандарт не накладывает ограничений на реализацию выполнения данной программы с данными входными параметрами (даже в отношении операций, которые предшествуют операции с неопределенным поведением).

6 Экземпляр каждого объекта с автоматическим временем жизни (6.7.3) связан с каждой точкой входа в его блок [Прим. пер.: Блок – часть кода исполняемой программы заключённая в фигурные скобки]. Такой объект существует и хранит последнее сохраненное значение пока блок выполняется или приостановлен (посредством вызова функции или получении сигнала).

7 Список требований к соответствующим реализациям:

(7.1) – Обращения через изменяемые glvalues [Прим. пер.: glvalue - сокращение от global value] вычисляются строго по правилам абстрактной машины.

(7.2) – При завершении программы все данные, записанные в файлы, должны быть идентичны одному из возможных результатов, который был бы получен в результате выполнении программы в соответствии с абстрактной семантикой.

(7.3) – Ввод и вывод в интерактивных устройствах должны быть реализованы таким образом, чтобы вывод запроса фактически осуществлялся до того, как программа начнет ожидание ввода. Что представляет собой интерактивное устройства определяется реализацией. В совокупности это называется наблюдаемым поведение программы. [Прим.: Более строгие соответствия между абстрактной и фактической семантиками определяются каждой из реализаций самостоятельно. – конец прим.]

8 [Прим.: Операторы могут быть разгруппированы в соответствии с обычными математическими правилами только там, где эти операторы действительно проявляют ассоциативные или коммутативные свойства. Например, в приведенном ниже фрагменте

int a, b;

/\* ... \*/

a = a + 32760 + b + 5;

выражение ведет себя также, как

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

из-за свойств ассоциативности и старшинства операторов. Таким образом, результат суммы (a + 32760) складывается с b, полученный результат складывается с 5, и, далее, значение результата выражения присваивается переменной a. На машине, где переполнения вызывают исключение и в которой диапазон значений для типа int равен [-32768, +32767], реализация не может изменить данное выражение на

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

так как если бы значения a и b были соответственно равны -32754 и -15, сумма a + b могла бы вызвать исключение, хотя оригинальное выражение исключений не вызывает; также выражение не может быть переписано как

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

или

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

так как значения a и b могут быть соответственно 4 и -8 или -17 и 12. Однако на машинах, где переполнение не вызывает исключение и на которых результат переполнения обратим, приведенное выше выражение может быть заменено реализацией на любое из приведенных, потому что будет получен один и тот же результат. – конец прим.]

9 Составное выражение определяется следующим образом:

(9.1) – Составным выражением выражения является это выражение.

(9.2) – Составными выражениями списка инициализации или (возможно заключенного в скобки) списка выражений являются составные выражения элементов соответствующего списка.

(9.3) – Составными выражениями инициализатора скобки или равно как формы секции инициализатора вида =, является составные выражения секции инициализатора.

[Пример:

struct A { int x; };

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

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

Составное выражение инициализатора использует для инициализации b и 5 и 1 + 1. – конец примера].

10 Непосредственными подвыражениями выражения e являются:

(10.1) – Составное выражение операндов e (Раздел 8).

(10.2) – Любой вызов функции в котором неявно вызывается e

(10.3) – Если e – лямбда-выражение (8.1.5), то подвыражениями e являются инициализация внутренних переменных, захваченных посредством копирования и составные выражения инициализатора при инициализирующем-захвате

(10.4) – Если е – вызов функции или неявный вызов функции (8.2.2), то подвыражениями е являются составные выражения каждого аргумента по умолчанию (11.3.6), которые используются в вызове или

(10.5) – Если е – создает агрегированный объект (11.6.1), подвыражениями е являются составные выражения для инициализаторов каждого из членов такого объекта, определенных по умолчанию.

11 Подвыражение выражения е является непосредственным подвыражением е или подвыражением непосредственного подвыражения е. [Прим.: Выражения, появляющиеся в составном операторе, лямбда-выражений не являются подвыражениями лямбда-выражений. – конец прим.]

12 Полное выражение это:

(12.1) – Не рассчитываемый операнд (Раздел 8).

(12.2) – Константное выражение (8.20).

(12.3) – Декларатор инициализации (Раздел 11) или инициализатор памяти (15.6.2), включая составное выражение инициализатора.

(12.4) – Вызов деструктора, сгенерированный в конце жизни объекта отличного от временного объекта (15.2) или

(12.5) – Выражение, которое не является подвыражением другого выражения и которое в ином случае не является частью другого полного выражения. Если конструкция языка определяется для неявного вызова функции, использование такой конструкции считается выражением для этого определения. Преобразования, применяемые к результату выражения для удовлетворения требований конструкции языка в которой выражение появляется, также считается частью полного выражения. Для инициализатора, выполняющего инициализацию сущности (включая расчет стандартных инициализаторов членов совокупности) также является частью полного выражения.

[Пример:

struct S {

S(int i): I(i) { } // полное выражение - инициализация I

int& v() { return I; }

~S() noexcept(false) { }

private:

int I;

};

S s1(1); // полное выражение – вызов S::S(int)

void f() {

S s2 = 2; // полное выражение – вызов of S::S(int)

if (S(3).v()) // полное выражение включает в себя lvalue-to-rvalue и

// int-to-bool преобразования и выполняется до

// удаления временного объекта в конце полного выражения

{ }

bool b = noexcept(S()); // спецификация исключения деструктора S

// рассматривается для noexept

// полное выражение – удаление s2 в конце блока

}

struct B {

B(S = S(0));

};

B b[2] = { B(), B() }; // полное выражение – внутренняя инициализация

// включая удаление временных объектов

- конец примера]

13 [Прим.: Расчет полного выражения может включать расчет подвыражений, которые не являются лексическими частями полного выражения. Например, подвыражения, участвующие в расчете аргументов по умолчанию (11.3.6) считаются созданными, в выражении, которое вызывает функцию, а не в выражении, которое определяет аргумент по умолчанию. – конец прим.]

14 Считывание объекта, помеченного как изменяемое glvalue (6.10), модификация объекта, вызов библиотечной функции ввода/вывода или вызов функции, которая выполняет какую-либо из этих операций вызывает побочный эффект, который является изменением в состоянии среды выполнения. Расчет выражения (или подвыражения) в общем случае включает и вычисление значения (включая определение идентичности объекта для расчета glvalue и извлечение значения, которое было присвоено ему ранее для расчета prvalue [Прим. пер. prvalue - сокращение от private value]) и создание побочных эффектов. Когда возвращается вызов библиотечной функции ввода/вывода или рассчитывается доступ через изменяемое glvalue, побочный эффект считается законченным, даже когда какие-либо внешние действия, которые подразумеваются в вызове (такие как сам ввод/вывод) или изменяемый доступ еще не завершены.

15 Предшествование в последовательности представляет собой асимметричную, переходную, попарную связь между расчетами, выполняемыми в одном потоке (4.7), которая включает частичную упорядоченность этих расчетов. Для двух любых расчетов А и В, если А предшествует В (или, что тоже самое, В следует за А), то выполнение А должно предшествовать выполнению В. Если А не предшествует В и В не предшествует А, тогда А и В не упорядочены [Прим.: Выполнение неупорядоченных расчетов может вызвать бесконечный цикл - конец прим.] Расчеты А и В неопределенно упорядочены, когда и А предшествует В или В предшествует А, но это не указано явно. [Прим.: Неопределенная упорядоченность расчётов не может привести к бесконечному циклу, и любой из расчётов может быть выполнен первым - конец прим.] Выражение X предшествует выражению Y, если все вычисления значений и все побочные эффекты, которые вызывает выражение X, предшествуют всем вычислениям значений и всем побочным эффектам, которые вызывает выражение Y.

16 Каждое вычисление значения и побочный эффект, ассоциированный с полным выражением, предшествует каждому вычислению и побочному эффекту, ассоциированному со следующим полным выражением, которое будет вычислено.

17 За исключением случаем, когда это явно указано, вычисление операндов индивидуальных операторов и подвыражений индивидуальных выражений неупорядоченно [Прим.: В выражениях, которые вычисляются более одного раза в течении вызова программы, неупорядоченные вычисления и вычисления с неопределенной упорядоченностью в их подвыражениях не обязательно должны выполняться в разных вычислениях - конец прим.] Вычисления значений операндов оператора предшествуют вычислению значения результата выполнения оператора. Если побочный эффект на участке памяти (4.4) имеет неупорядоченное отношение к другому побочному эффекту на том же участке памяти или вычисление значения использует значение какого-либо объекта из того же участка памяти, и они потенциально не выполняются параллельно – такое поведение не определено [Прим.: Следующий раздел налагает аналогичные, но более общие ограничения на потенциальные параллельные вычисления - конец прим.]

[Пример:

void g(int i) {

i = 7, i++, i++; // i стало равно 9

i = i++ + 1; // значение i не определенно

i = i++ + i; // неопределенное поведение

i = i + 1; // значение i не определено

}

- конец примера]

18 При вызове функции (не важно является ли функция inline), любое вычисление значения и побочный эффект, ассоциированный с любым выражением в аргументе функции или с постфиксным выражением, обозначающим вызов функции, предшествует выполнению любого выражения или оператора в теле вызываемой функции. Для каждого вызова функции F, для каждого расчета A, который происходит внутри F и для каждого расчета B, который не происходит внутри F, но выполняется в том же потоке и как часть того же обработчика сигналов (если такой имеется), A предшествует B или B предшествует A. [Прим.: Если с другой стороны A и В могут быть не упорядочены, тогда они имеют неопределенную упорядоченность, - конец прим.] В некоторых контекстах в C++ вычисления вызовов функций происходят даже если в единице трансляции нет соответствующего синтаксиса вызова функции [Например: Вычисление new-выражения вызывает одну или более функцию выделения памяти или конструктор; см. 8.3.4. Другой пример – вызов функции преобразования (15.3.2) может возникать в контексте, в котором нет синтаксиса вызова функции – конец примера] Ограничения в упорядочивании исполнения вызовов (в описании выше) являются особенностями вызовов функций как вычислений, каким бы ни был синтаксис выражений где эти функции вызываются.

19 Если обработчик сигналов исполняется как результат вызова функции std::raise, тогда исполнение обработчика происходит после вызова функции std::raise и до возврата из нее [Прим.: Когда сигнал получен по другим причинам, исполнение обработчика сигналов обычно не упорядочено по отношению к остальной части программы - конец прим.]