КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 05. Выражения и операторы языка С

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)

+375 17 320 7402 (ОИПИ НАНБ)

prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by/

Кафедра ЭВМ, 2021

2028.09.29 (14:00-14:40)

Оглавление

Взаимодействие и правила поведения на занятиях

Язык общения — русский
Фото- и видеосъемка запрещается, болтовня и прочее мычание тоже
Использование мобильных гаджетов может вызвать проблемы

prep@lsi.bas-net.by
ftp://student@lsi.bas-net.by/

Старосты групп отправляют на **prep@** со своего личного ящика сообщение, в котором указывают свой телефон и ящик, к которому имеют доступ все студенты группы. В этом же сообщении в виде вложения приводят списки своих групп (**.odt** или **.doc**). Если у группы нет ящика, его следует создать. Все студенты группы должны мониторить этот ящик.

Формат темы этого сообщения: «**010901 Фамилия И.О. Список группы**»
Формат темы для **prep@** вообще: «**010901 Фамилия И.О. Суть сообщения**»

Правила составления сообщений
- текстовый формат сообщений;

Удаляется на сервере присланное в ящик prep@ все, что:
- без темы;
- имеет тему не в формате;
- содержит html, xml и прочий мусор;
- содержит рекламу, в том числе и сигнатуры web-mail серверов;
- содержит ссылки на облака и прочие гуглопомойки вместо прямых вложений.

Выражения

Выражение — это последовательность операторов и операндов, которая:

- задает вычисление значения;
- обозначает объект;
- обозначает функцию;
- генерирует побочные эффекты;
- выполняет комбинацию вышеперечисленного.

Вычисление значений операндов оператора выполняется до вычисления значения результата оператора.

Поведение выражения не определено:

- если не определена последовательность действий побочного эффекта на скалярный объект относительно другого побочного эффекта на тот же скалярный объект;
- если не определена последовательность действий побочного эффекта на скалярный объект относительно вычисления значения с использованием значения того же скалярного объекта.
- если существует несколько допустимых порядков следования подвыражений в выражении и если такой неупорядоченный побочный эффект возникает в каком-либо из порядков следования.

Группировка операторов и операндов при вычислении выражения выполняется согласно приоритету операторов:

0) первичное-выражение, в частности, заключенное в скобки выражение вида **(** *выражение* **)**
1) оператор доступа к массиву по индексу **[]**;
2) оператор вызова функции **()**;
3) оператор доступа к членам структуры или объединения **.** или **->** ;
4) постфиксные операторы инкремента и декремента **++** и **--**;
5) составные литералы **(** *имя типа* **) {** *список-инициализаторов* **}**;
6) префиксные операторы инкремента и декремента **++** и **--**;
7) операторы получения адреса и косвенного доступа к объекту **&** и *****;
8) унарные арифметические операторы **+ - ~ !**;
9) операторы **sizeof** и **sizeof (** *имя-типа* **)**;
10) оператор **_Alignof**;
11) операторы приведения типа;
12) мультипликативные операторы *** / %**;
13) аддитивные операторы **+ -**;
14) операторы побитового сдвига **<< >>**;
15) операторы отношений **< > <= >=**;
16) операторы равенства **== !=**;
17) побитовый оператор И **&**;
18) побитовый оператор исключающего ИЛИ **^**;
19) побитовый оператор включающего ИЛИ **|**;

20) логический оператор И **&&**;
21) логический оператор ИЛИ **||**;
22) условный оператор **? :** ;
23) оператор простого присваивания **=** ;
24) операторы составного присваивания ***= /= %= += -= <<= >>= &= ^= |=** ;
25) оператор **запятая , .**

Порядок действия побочных эффектов и вычисления значений подвыражений обычно не устанавливается. Но есть исключения и они будут указаны далее.

Классический пример копирования последовательности байт

```
char *ss; // ss – указатель на область памяти откуда копировать
char *dd; // dd – указатель на область памяти куда копировать
...
size_t cnt = bytes2copy;  // размер копируемого блока данных в байтах
while (cnt-- > 0) {
    *dd++ = *ss++; // а как же приоритеты * и постфиксоного ++?
}
```

Некоторые операторы, например унарный оператор **~** и бинарные (побитовые) операторы **<<, >>, &, ^** и **|** должны иметь операнды целочисленного типа.

Эти операторы выдают значения, которые зависят от внутренних представлений целых чисел, и имеют определяемые реализацией и неопределенные результаты для знаковых типов.

Lvalue и rvalue[1]

Lvalue (locator value) — это выражение, имеющее тип, отличный от **`void`**, которое *потенциально* представляет собой объект, который занимает идентифицируемое место в памяти (например, имеет адрес).

Название «**lvalue**» происходит от выражения присваивания **`E1 = E2`**, в котором левый операнд **`E1`** должен быть (модифицируемым) **lvalue**. Другими словами,

> **lvalue** — это то, что может появиться слева от оператора присваивания.

rvalue определяется путём исключения в том смысле, что любое выражение является либо **lvalue**, либо **rvalue**. Таким образом из определения **lvalue** следует, что **rvalue** — это выражение, которое не представляет собой объект, который занимает идентифицируемое место в памяти. Обычно термином «**rvalue**» обозначают «значение выражения».

Очевидным примером **lvalue** является идентификатор объекта.

Изменяемое **lvalue** (modifiable lvalue) — это **lvalue**,

- тип которого не является незавершенным типом массива;
- не является константным типом;
- если это структура или объединение, не имеет какого-либо члена с типом, определенным как **`const`** (включая, рекурсивно, любой член или элемент всех содержащихся агрегатов или объединений).

---

1 На хабре есть статья «Понимание lvalue и rvalue в C и C++» — https://habr.com/ru/post/348198/

Примеры

```
int var; // объект с идентифицируемым местом в памяти
var = 4; // и ему можно присвоить значение
```

Оператор присваивания ожидает **lvalue** с левой стороны, и **var** является **lvalue**, потому что это объект с идентифицируемым местом в памяти.

С другой стороны, следующее приведет к ошибкам:

```
4 = var;       // ERROR! Константа
(var + 1) = 4; // ERROR! (var + 1) не связано ни с каким местом в памяти
```

Ни константа 4, ни выражение **var + 1** не являются **lvalue**, что автоматически их делает **rvalue**.

Они не **lvalue**, потому что оба являются временным результатом выражений, которые не имеют определённого места в памяти (то есть они могут находится в каких-нибудь временных регистрах на время вычислений). Поэтому присваивание в данном случае не несёт в себе никакого семантического смысла. Иными словами — некуда присваивать.

```
int *foo();
int *p;
*foo() = var;   // нормально
*(p + 1) = var; // нормально
```

Эффективный тип объекта — тип объекта, используемый для доступа к его сохраненному значению (объявленный тип объекта, если таковой имеется).

Термины **lvalue** и **rvalue** являются базовыми понятиями, лежащими в основе языков C и C++, хотя с ними практически не приходится сталкиваться при программировании.

Наиболее вероятное место столкнуться с ними — это сообщения компилятора. Например, при компиляции следующего кода компилятором **gcc**:

```
int foo() { return 2; }
int main() {
    foo() = 2;
    return 0;
}
```

Мы получим нечто следующее:

```
test.c: In function 'main':
test.c:8:5: error: lvalue required as left operand of assignment
```

Другой пример при компиляции следующего кода при помощи g++:

```
int& foo() {  // функция определена, как возвращающая ссылку на int
    return 2; // а возвращает int
}
```

Мы увидим:

```
testcpp.cpp: In function 'int& foo()':
testcpp.cpp:5:12: error: invalid initialization of non-const reference
of type 'int&' from an rvalue of type 'int'
```

Первичные выражения (наивысший приоритет вычисления)

первичное-выражение :

идентификатор
константа
строковый-литерал
(*выражение* **)**

Семантика

Идентификатор является *первичным* выражением при условии, что он объявлен:
1) как обозначающий объект, и в этом случае он является **lvalue**;
2) как обозначающий функцию (в этом случае это указатель функции).

Константа так же является первичным выражением. Тип константы зависит от ее формы и значения.

Строковый литерал так же является первичным выражением.

Заключенное в скобки выражение так же является первичным выражением. Его тип и значение идентичны типу выражения без скобок.

Заключенное в скобки выражение может являться
- **lvalue;**
- обозначением функции;
- **`void`**-выражением,

если *не заключенное в скобки выражение* является **lvalue**, указателем функции или **`void`**-выражением, соответственно.

Постфиксные операторы

постфикс-выражение :

постфикс-выражение	
постфикс-выражение **[ ]**	массивы
постфикс-выражение **[** *выражение* **]**	массивы
постфикс-выражение **( )**	вызов функции
постфикс-выражение **(** *выражение-список-аргументов* **)**	вызов функции
постфикс-выражение **.** *идентификатор*	доступ к члену
постфикс-выражение **->** *идентификатор*	доступ к члену
постфикс-выражение **++**	постфиксный инкремент
постфикс-выражение **--**	постфиксный декремент
(*имя-типа* **) {** *список-инициализаторов* **}**	составной литерал
(*имя-типа* **) {** *список-инициализаторов* **, }**	составной литерал

выражение-список-аргументов:

выражение-присваивания

выражение-список-аргументов, *выражение-присваивания*

Оператор индексации массива

постфикс-выражение :
постфикс-выражение **[** *expression* **]**

Семантика

Постфикс-выражение, за которым следует *выражение* в квадратных скобках **[ ]**, является *индексным* обозначением элемента объекта массива.

Оператор индекса **[]** определяется таким образом, что конструкция **E1[E2]** идентична конструкции **(*((E1)+(E2)))**.

Из этого следует, что в языках C и C++, массивов, как сущностей, нет — имя **E1** обозначает не сам массив, а указатель на начальный элемент массива. Тем не менее, такой тип, как тип массива существует. Индексация элементов массива начинается с нуля.

Следующие друг за другом операторы индекса обозначают многомерный массив.

```
long int array[slices][rows][cols]; // hyperspectral data cube
```

Правило

Если **E** — это n-мерный массив $(n \geqslant 2)$ с размерами $i \times j \times \cdots \times k$, то **E** преобразуется в указатель на $(n-1)$-мерный массив с размерами $j \times \cdots \times k$.

Если к этому указателю явно или неявно применяется унарный оператор * (в результате индексации), результатом является $(n-1)$-мерный массив, который сам преобразуется в указатель, если он используется не как **lvalue**. Из этого следует, что

Массивы хранятся в основном порядке строк
(последний индекс изменяется быстрее всего).

Примеры

Рассмотрим объект, определенный объявлением

```
int x[3][5];
```

Здесь **x** — массив целых размерности 3 x 5. Точнее говоря, **x** — это массив трехэлементных объектов, каждый из которых представляет собой массив из пяти целых.

В выражении **x[i]**, которое эквивалентно **(*((x)+(i)))**, **x** сначала преобразуется в указатель на начальный массив из пяти целых чисел.

Затем **i** корректируется в соответствии с типом **x**, что концептуально влечет за собой умножение **i** на размер объекта, на который указывает указатель, а именно массив из пяти **int**-объектов.

Результаты складываются и чтобы получить массив из пяти целых, применяется операция косвенного доступа * (операция получения объекта по указателю, разадресация указателя).

При использовании в выражении **x[i][j]** этот массив, в свою очередь, преобразуется в указатель на первое из целых чисел, поэтому **x[i][j]** возвращает **int**.

Вызовы функций

Синтаксис

вызов функции:

постфикс-выражение **( )**

постфикс-выражение **(** *выражение-список-аргументов* **)**

выражение-список-аргументов:

выражение-присваивания

выражение-список-аргументов **,** *выражение-присваивания*

Семантика

Постфикс-выражение, за которым следуют скобки **( )**, содержащие, возможно пустой, список выражений, разделенных запятой, является вызовом функции.

```
csin(z);                    // double complex z;
fprintf(fd, "%s\n", name);  // char *name;
```

постфикс-выражение обозначает вызываемую функцию.

выражение-список-аргументов указывает аргументы функции.

Аргумент может быть выражением любого полного (завершенного) объектного типа. Однако,

Параметр, объявленный как имеющий *тип массива* или имеющий *тип функции*, корректируется таким образом, чтобы он имел тип указателя.

С другой стороны, можно передать указатель на объект, и функция может изменить значение объекта, на который ссылается указатель.

При подготовке к вызову функции аргументы вычисляются, и каждому параметру присваивается значение соответствующего аргумента.

Функция может изменять значения своих параметров, но эти изменения не могут влиять на значения аргументов (область видимости аргументов функции — тело функции).

Если выражение, обозначающее вызываемую функцию, имеет тип «указатель на функцию, которая возвращает объект некоторого типа», выражение вызова функции имеет тот же тип, что и этот тип объекта.

В противном случае вызов функции имеет тип **`void`**.

Допускаются рекурсивные вызовы функций как прямо, так и косвенно через любую последовательность других функций.

Ограничения (о)

Выражение, которое обозначает вызываемую функцию[2], должно иметь тип «указатель на функцию, возвращающую **`void`**» или «указатель на функцию, возвращающую полный тип объекта, отличный от типа массива».

Если выражение, которое обозначает вызываемую функцию, имеет тип, который включает в себя прототип, количество аргументов в прототипе должно совпадать с количеством параметров.

Каждый аргумент должен иметь такой тип, чтобы его значение могло быть присвоено объекту, имеющему неквалифицированную версию того типа, который имеет соответствующий ему параметр.

---

2 Чаще всего это результат преобразования идентификатора, который является обозначением функции.

Пример

В вызове функции

```
(*pf[f1()])(f2(), f3() + f4()) // постфикс-выражение ( выражение-список-аргументов )
```

функции **f1**, **f2**, **f3** и **f4** могут вызываться в любом порядке, тем не менее, все сторонние эффекты будут завершены до вызова функции, на которую указывает **pf[f1()]**.

Члены структур и объединений

Синтаксис

член-структуры-или-объединения:

постфикс-выражение **.** *идентификатор*

постфикс-выражение **->** *идентификатор*

Семантика

Постфиксное выражение, за которым следует оператор **.** и идентификатор, обозначают член структуры или объединения.

Значение выражения соответствует значению данного члена и является **lvalue**, если первое выражение является **lvalue**. Если первое выражение имеет уточненный тип, результат будет иметь уточненную версию типа указанного члена.

Постфиксное выражение, за которым следует оператор **->** и идентификатор, обозначает член структуры или объединения. Оно имеет значение именованного члена объекта, на которое указывает первое выражение, и является **lvalue**.

Если **&E** является допустимым выражением указателя (где **&** является оператором получения адреса объекта), который генерирует указатель на свой операнд), выражение **(&E)->MOS** совпадает с **E.MOS**.

Ограничения

Первый операнд оператора точка `.` должен иметь атомарный, квалифицированный или неквалифицированный тип структуры или атомарный, квалифицированный или неквалифицированный тип объединения, а второй операнд должен называть член данного типа.

Первый операнд оператора `->` должен иметь тип «указатель на атомарную, квалифицированную или неквалифицированную структуру» или «указатель на атомарное, квалифицированное или неквалифицированное объединение», а второй операнд должен называть член данного типа.

Пример 1.

Если **f** — это функция, возвращающая структуру или объединение, а **X** — это член соответствующей структуры или объединения, то **f().X** является допустимым постфиксным выражением, но не является **lvalue**.

Пример 2.

Уточненный тип при определении переносится на тип члена

```
struct s {
  int i;
  const int ci;
};
struct s s;
const struct s cs;    // уточняем тип структуры при ее определении
volatile struct s vs; // уточняем тип структуры при ее определении
```

различные члены имеют типы:

```
s.i    int
s.ci   const int
cs.i   const int
cs.ci  const int
vs.i   volatile int
vs.ci  volatile const int
```

Операторы постфиксного инкрементирования и постфиксного декрементирования

Синтаксис

постфикс-выражение **++**
постфикс-выражение **--**

Семантика

Результатом постфиксного оператора ++ является значение операнда. Увеличение значения операнда (к нему добавляется значение 1 соответствующего типа) является сторонним эффектом.

Вычисление значения результата всегда следует **перед** сторонним эффектом обновления сохраненного значения операнда.

Классика пересылки строк (K&R):

```
char *ss, *dd;
...
while((*dd++ = *ss++) != '\0');
```

Постфиксный оператор **--** аналогичен постфиксному оператору **++**, за исключением того, что значение операнда уменьшается (т.е. из него вычитается значение **1** соответствующего типа).

Ограничения

Операнд постфиксного оператора увеличения или уменьшения должен иметь атомарный, квалифицированный или неквалифицированный вещественный тип или тип указателя, а также должен быть модифицируемым **lvalue**.

Составные литералы

Синтаксис

(*имя-типа* **) { ***список-инициализаторов* **}**
(*имя-типа* **) { ***список-инициализаторов* **, }**

Семантика

Постфиксное выражение, которое состоит из имени типа в скобках, за которым следует заключенный в скобки список инициализаторов, является *составным литералом*.

Он предоставляет ***неименованный*** объект, значение которого задается списком инициализации[3].

Если имя типа указывает на массив неизвестного размера, размер определяется списком инициализации, а типом составного литерала будет являться завершенный тип массива. В противном случае (когда имя типа указывает тип объекта), тип составного литерала соответствует типу, указанному в имени типа. В любом случае результатом является **lvalue**.

Значением составного литерала является значение безымянного объекта, инициализированного списком инициализации.

Если составной литерал находится вне тела функции, объект имеет статическую продолжительность хранения; в противном случае он имеет автоматическую продолжительность хранения, связанную с окружающим его блоком.

К составным литералам применяются все семантические правила для списков инициализации, в частности, подобъекты без явных инициализаторов инициализируются нулем.

---

3 Следует обратить внимание, что это выражение отличается от выражения приведения. Например, приведение определяет преобразование только в скалярные типы или только void, при этом результат выражения приведения не является lvalue.

Ограничения

Имя типа должно указывать полный тип объекта или массив неизвестного размера, но не тип массива переменной длины.

Пример 1.

Определение с областью видимости области файла

```
int *p = (int []){2, 4};
```

инициализирует указатель **p** таким образом, чтобы тот указывал на первый элемент массива из двух целых, первый из которых имеет значение два, а второй — четыре.

Выражения в этом составном литерале должны быть постоянными.

Безымянный объект имеет статическую продолжительность хранения.

Пример 2.

В противоположность предыдущему случаю

```
void f(void) {
    int *p;
    /*...*/
    p = (int [2]){*p};
    /*...*/
}
```

указателю **p** назначается адрес первого элемента массива из двух целых, первый из которых имеет значение, на которое ранее указывал **p**, а второй — ноль.

Выражения в этом составном литерале не обязательно должны быть постоянными.

Безымянный объект в этом случае имеет автоматическую продолжительность хранения.

Пример 3.

Инициализаторы с именами можно комбинировать с составными литералами.

Объекты структуры, созданные с использованием составных литералов, могут передаваться в функции независимо от порядка членов:

```
drawline((struct point){.x=1, .y=1},
         (struct point){.x=3, .y=4});
```

Или, если **drawline** вместо структур ожидает указатели на **struct point**:

```
drawline(&(struct point){.x=1, .y=1},
         &(struct point){.x=3, .y=4});
```

Пример 4.

Составной литерал только для чтения может быть указан с помощью конструкций, типа:

(const float []){1e0, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6}

Пример 5.
Следующие три выражения имеют разные значения (meanings):

```
"/tmp/fileXXXXXX"                   // static не изменяемое
(char []){"/tmp/fileXXXXXX"}        // auto изменяемое
(const char []){"/tmp/fileXXXXXX"} // auto не изменяемое
```

Первый всегда имеет статическую длительность хранения и имеет тип массив **char**, и должен быть неизменяемым.

Последние два имеют автоматическую длительность хранения, если они встречаются в теле функции, при этом первое из них является изменяемым.

Пример 6.
Как и строковые литералы, составные литералы с квалификацией **const** могут быть помещены в постоянную память и даже могут использоваться совместно. Например, выражение

```
(const char []){"abc"} == "abc"
```

может выдать 1, если память литералов используется совместно.

Пример 7.

Поскольку составные литералы не имеют имени, один составной литерал не может указывать объекты с циклической связью. Например, нет способа написать самоссылающийся составной литерал, который мог бы использоваться в качестве аргумента функции вместо именованного объекта **`endless_zeros`**, приведенного ниже:

```
struct int_list {
    int car;
    struct int_list *cdr;
};
struct int_list endless_zeros = {0, &endless_zeros};
eval(endless_zeros);
```

Пример 8.

Каждый составной литерал создает только один объект в данной области видимости:

```
struct s { int i; };
int f(void) {
    struct s *p = 0, *q;
    int j = 0;
again:
    q = p, p = &((struct s){ j++ });
    if (j < 2) {
        goto again;
    }
    return p == q && q->i == 1;
}
```

Функция **f()** всегда возвращает значение **1**.

Следует обратить внимание, что если бы вместо явного оператора **goto** и оператора метки, использовался итерационный оператор, время жизни безымянного объекта ограничивалось бы только телом цикла, и при следующем входе в цикл **p** мог бы иметь неопределенное значение, что привело бы к неопределенному поведению.

Унарные операторы

Синтаксис

унарное-выражение:

постфикс-выражение

++ *унарное-выражение* оператор префиксного инкрементирования

-- *унарное-выражение* оператор префиксного декрементирования

унарный-оператор cast-выражение

sizeof *унарное-выражение*

sizeof (*имя-типа* **)**

_Alignof (*имя-типа* **)**

унарный-оператор: один из

& * + - ~ !

Операторы префиксного инкрементирования и декрементирования

Синтаксис

++ *унарное-выражение*	оператор префиксного инкрементирования
-- *унарное-выражение*	оператор префиксного декрементирования

Семантика

Значение операнда префиксного оператора **++** увеличивается.

Результатом является новое значение операнда после увеличения

Выражение **++E** эквивалентно (**E += 1**).

Префиксный оператор **--** аналогичен префиксному оператору **++**, за исключением того, что значение операнда уменьшается.

Ограничения

Операнд префиксного оператора увеличения или уменьшения должен иметь атомарный, квалифицированный или неквалифицированный тип или тип указателя тип и должен быть модифицируемым lvalue.

Операторы получения адреса и разадресации/разыменования

Адрес

Унарный оператор **&** выдает адрес своего операнда. Если операнд имеет тип «**type**», результат применения данного оператора будет иметь тип «**указатель на type**».

Если операнд является результатом применения унарного оператора разадресации * (**&*p**), то ни этот оператор, ни оператор **&** не вычисляются, а результатом является, как если бы оба были опущены, за исключением того, что все еще применяются ограничения на операторы, и результат не является **lvalue**.

Аналогично, если операнд является результатом оператора **[]**, (**&array[idx]**) ни оператор **&**, ни оператор **[]**, ни подразумеваемый унарный * не вычисляются, а результатом является, как если бы оператор **&** был удален, а оператор **[]** был изменен на оператор **+** (**array+idx**).

В противном случае результатом является указатель на объект или функцию, обозначенный его операндом.

Указатель

Унарный оператор * обозначает косвенную адресацию.

Если операнд является указателем на функцию, результатом является обозначение/имя функции.

Если он указывает на объект, результатом является **lvalue**, обозначающее объект.

Если операнд имеет тип «**указатель на T**», результат будет иметь тип «**T**».

Недопустимые значения для разыменования указателя с помощью унарного оператора ***:**

- нулевой указатель;
- адрес, неправильно выровненный для типа объекта, на который указывает указатель;
- адрес объекта после окончания его срока действия.

Если указателю было присвоено недопустимое значение, поведение оператора ***** не определено.

Выражение **&*E** эквивалентно **E** (даже если **E** является нулевым указателем), а **&(E1[E2])** эквивалентно **((E1) + (E2))**.

Всегда верно, что если **E** является обозначением/именем функции или **lvalue**, который является допустимым операндом унарного оператора **&**, то ***&E** является обозначением функции или **lvalue**, равным **E.**

Если ***P** — это **lvalue**, а **T** — имя объекта с типом «**указатель на объект**», то ***(T)P** является **lvalue**, тип которого совместим с типом, на который указывает **T**.

Ограничения

Операнд унарного оператора **&** должен быть одним из:

- имя функции;
- результат оператора доступа по индексу **[]**;
- результат унарного оператора косвенной адресации *****;
- **lvalue**, которое обозначает объект, не являющийся битовым полем и не объявленный со спецификатором класса хранения **register**.

Операнд унарного оператора ***** должен иметь тип указателя.

Унарные арифметические операторы

Синтаксис

```
+ - ~ !
```

Семантика

Результатом унарного оператора **+** является значение его операнда.

Результат унарного оператора **-** является отрицательным для его операнда.

Результат оператора **~** является побитовым дополнением его операнда (то есть каждый бит в результате устанавливается тогда и только тогда, когда соответствующий бит в расширенном по типу операнде не установлен).

Над операндом выполняются целочисленные распространения типа ранга, и результат имеет распространенный тип (тип повышенного ранга). Если распространенный тип является беззнаковым типом, выражение **~E** эквивалентно максимальному значению, представляемому в этом типе, минус **E**.

Результат оператора логического отрицания **!** равен 0, если значение его операнда не равно 0, и 1, если значение его операнда равно 0. Результат имеет тип **int**.

Выражение **!E** эквивалентно **(0 == E)**.

Ограничения

Операнд унарного оператора **+** или **-** должен иметь арифметический тип.

Операнд унарного оператора **~** должен иметь целочисленный тип.

Операнд унарного оператора **!** должен иметь скалярный тип.

Операторы sizeof и _Alignof

Синтаксис

`sizeof` *унарное-выражение*
`sizeof ( ` *имя-типа* **` )`**
`_Alignof ( ` *имя-типа* **` )`**

Семантика

Оператор **`sizeof`** возвращает размер в байтах своего операнда, который может быть выражением или именем типа в скобках.

Размер определяется по типу операнда. Результатом является целое число.

Если тип операнда является типом массива переменной длины, операнд вычисляется по месту, в противном случае, операнд не вычисляется, а результатом является целочисленная константа.

Оператор **`_Alignof`** выдает требование выравнивания согласно типу своего операнда. Операнд не вычисляется, а результатом является целочисленная константа.

При применении к типу массива результатом является требование выравнивания для типа элемента.

sizeof

Когда к операнду с типом **char**, **unsigned char** или **signed char** (или их квалифицированной версией) применяется оператор **sizeof** результат равен 1.

При применении к операнду с типом массива результатом является общее число байтов в массиве.

При применении к параметру функции, объявленному как имеющий тип массива или тип функции, оператор **sizeof** дает размер скорректированного типа (размер указателя).

При применении к операнду, который имеет тип структуры или тип объединения, результатом является общее количество байтов в таком объекте, включая внутреннее и конечное (padding) заполнение.

Значение результата обоих операторов определяется реализацией, а его тип (целочисленный тип без знака) — **size_t**, определенный в **<stddef.h>** (и других заголовках).

Ограничения

Оператор **sizeof** не должен применяться к выражению, имеющему тип функции или неполный тип, к имени такого типа в скобках или к выражению, которое обозначает член битового поля. Оператор **_Alignof** не должен применяться к типу функции или неполному типу.

Пример 1.

Основное использование оператора **sizeof** связано с процедурами, такими как распределители (аллокаторы) памяти и системы ввода/вывода. Функция выделения памяти может принимать размер (в байтах) объекта для выделения и возвращать указатель на **void**. Например:

```
extern void *alloc(size_t);
double *dp = alloc(sizeof *dp);
```

Реализация функции **alloc** должна гарантировать, что ее возвращаемое значение выровнено соответствующим образом для преобразования в указатель на **double**.

Пример 2.

Другое использование оператора **sizeof** — вычисление количества элементов в массиве:

```
sizeof array/sizeof array[0]
```

Пример 3.

В этом примере вычисляется размер массива переменной длины и возвращается из функции:

```
#include <stddef.h>

size_t fsize3(int n) {

    char b[n+3];      // VLA – масси переменной длины
    return sizeof b;  // sizeof вычисляется во время выполнения
}

int main() {

    size_t size;

    size = fsize3(10); // fsize3 returns 13
    return 0;
}
```

Операторы приведения типа

Синтаксис

cast-выражение:

унарное-выражение

(*имя-типа* **)** *cast-выражение*

Семантика

Имя типа в скобках, предшествующее выражению, преобразует значение этого выражения к указанному типу.

Эта конструкция называется приведением типов. Приведение, которое не указывает преобразование, не имеет никакого эффекта на тип или значение выражения.

Если значение выражения представлено с большим диапазоном или точностью, чем требуется для типа, указанного оператором приведения, то приведение усекает любой лишний диапазон и точность.

Ограничения

Если в имени типа не указан тип **`void`**, имя типа должно указывать атомарный, квалифицированный или неквалифицированный скалярный тип, а операнд должен иметь скалярный тип.

Тип указателя не должен преобразоваться ни в какой плавающий тип. Плавающий тип не должен преобразоваться ни в какой тип указателя.

Мультипликативные операторы

Синтаксис

мультипликативное-выражение:

cast-выражение

мультипликативное-выражение ***** *cast-выражение*

мультипликативное-выражение **/** *cast-выражение*

мультипликативное-выражение **%** *cast-выражение*

Семантика

Над операндами выполняются обычные арифметические преобразования.

Результатом бинарного оператора ***** является произведение операндов.

Результатом оператора **/** является частное от деления первого операнда на второй.

Результатом оператора **%** является остаток от деления первого операнда на второй.

Для операций **/** и **%**, если значение второго операнда равно нулю, поведение не определено.

При делении целых чисел результатом оператора **/** является алгебраическое частное с любой отброшенной дробной частью. Это часто называют «усечением в направлении нуля».

Ограничения

Каждый из операндов должен иметь арифметический тип.

Операнды оператора **%** должны иметь целочисленный тип.

```
ldiv_t ldiv(long p, long q); // вычисляет частное и остаток деления p на q
```

Аддитивные операторы

Синтаксис

аддитивное-выражение:

мультипликативное-выражение

аддитивное-выражение **+** *мультипликативное-выражение*

аддитивное-выражение **-** *мультипликативное-выражение*

Семантика

Если оба операнда имеют арифметический тип, то над ними выполняются обычные арифметические преобразования.

Результатом бинарного оператора **+** является сумма операндов.

Результатом работы бинарного оператора **-** является разность, возникающая в результате вычитания второго операнда из первого.

Арифметика указателей

Если выражение с целочисленным типом добавляется или вычитается из указателя, результат имеет тип операнда указателя.

Если операнд-указатель указывает на элемент объекта массива, и массив достаточно велик, результат указывает на смещение элемента от исходного элемента, так что разность индексов результирующего и исходного элементов массива равна целочисленному выражению.

Другими словами, если выражение **P** указывает на i-й элемент объекта массива, выражения **(P) + N** (эквивалентно, **N + (P)** и **(P) — N** (где **N** значение **n**) указывают, соответственно, на **(i+n)**-й и **(i-n)**-й элементы массива, если они существуют.

Кроме того, если выражение **P** указывает на последний элемент объекта массива, выражение **(P) + 1** указывает на элемент, следующий за последним элементом массива, а если выражение **Q** указывает на элемент, следующий за последним элементом массива, выражение **(Q) — 1** указывает на последний элемент массива.

Когда вычитаются два указателя, оба должны указывать на элементы одного и того же объекта массива или на элемент, следующий за последним элементом массива. Результатом такого вычитания является разность индексов двух элементов массива.

Размер результата определяется реализацией, а его тип (целочисленный тип со знаком) — типом **`ptrdiff_t`**, определенным в заголовке **`<stddef.h>`**.

Если результат не может быть представлен в объекте этого типа, поведение не определено.

Другими словами, если выражения **P** и **Q** указывают, соответственно, на i-й и j-й элементы объекта массива, выражение **(P) - (Q)**, если оно вписывается в объект типа **`ptrdiff_t`**, имеет значение **i - j**.

Более того, если выражение `P` указывает либо на элемент объекта массива, либо на элемент, следующий за последним элементом массива, а выражение `Q` указывает на последний элемент того же объекта массива, выражение `((Q)+1)-(P)` имеет то же значение, что и `((Q)-(P))+1` и `-((P)-((Q)+1))`.

Это же выражение имеет нулевое значение, если выражение `P` указывает на элемент, следующий за последним элементом массива, хотя выражение `(Q)+1` на элемент массива не указывает.

Есть еще один способ подхода к арифметике указателей — сначала преобразовать указатели в указатели на символ. В этой схеме целочисленное выражение, добавляемое или вычитаемое из преобразованного указателя, сначала умножается на размер первоначально указываемого объекта, а результирующий указатель преобразуется обратно в исходный тип. При вычитании указателя результирующая разность между указателями на символ следует разделить на размер объекта. С этой точки зрения, чтобы обеспечить возможность существования «элемента, следующего за последним» реализация должна предоставлять только один дополнительный байт (который может перекрывать другой объект в программе) сразу после конца объекта.

Ограничения

В случае сложения либо оба операнда должны иметь арифметический тип, либо один операнд должен быть указателем на полный тип объекта, а другой должен иметь целочисленный тип.

В случае вычитания должно выполняться одно из следующих условий:

- оба операнда имеют арифметический тип;
- оба операнда являются указателями на квалифицированные или неквалифицированные версии совместимых полных типов объектов;
- левый операнд является указателем на полный тип объекта, а правый операнд имеет целочисленный тип.

Инкремент (приращение) эквивалентно добавлению 1.

Декремент (уменьшение) эквивалентно вычитанию 1.

Пример

Арифметика указателей хорошо накладывается на указатели на типы массивов переменной длины.

```
{
    int n = 4;
    int m = 3;
    int a[n][m];
    int (*p)[m] = a; // p == &a[0]
    p += 1;          // p == &a[1]
    (*p)[2] = 99;    // a[1][2] == 99
    n = p - a;       // n == 1
}
```

Если бы массив **a** в вышеприведенном примере был объявлен как массив известного постоянного размера, а указатель **p** был объявлен как указатель на массив с таким же известным постоянным размером (указывающий на **a**), результаты были бы такими же.

Операторы побитового сдвига

Синтаксис

сдвиговое-выражение:

аддитивное-выражение

сдвиговое-выражение **<<** *аддитивное-выражение*

сдвиговое-выражение **>>** *аддитивное-выражение*

Ограничения

Каждый из оперндов должен иметь целочисленный тип

Семантика

Перед сдвигом над каждым из операндов выполняется целочисленное распространение типа (integer promotion). Тип результата — тип распространенного левого операнда.

Если значение правого операнда отрицательно или больше или равно ширине распространенного левого операнда, поведение не определено.

Результатом выражения **E1 << E2** является **E1**, сдвинутая влево на E2 битовых позиций, а освобождаемые биты заполняются нулями.

Если **E1** имеет беззнаковый тип, значение результата равно $\mathbf{E1 \times 2^{E2}}$, приведенное по модулю на единицу больше, чем максимальное значение, представляемое в типе результата.

Если **E1** имеет тип со знаком и неотрицательное значение, а $\mathbf{E1 \times 2^{E2}}$ представимо в типе результата, то это и будет результирующим значением, в противном случае поведение не определено.

Результатом выражения **E1 >> E2** является **E1**, сдвинутое вправо на **E2** битовых позиций. Если **E1** имеет беззнаковый тип, или если **E1** имеет тип со знаком и неотрицательное значение, значение результата является целой частью частного $\mathbf{E1/2^{E2}}$. Если **E1** имеет тип со знаком и отрицательное значение, результирующее значение определяется реализацией.

Реляционные операторы (Операторы отношений)

Синтаксис

реляционное-отношение:

сдвиговое-выражение

реляционное-отношение **<** *сдвиговое-выражение*

реляционное-отношение **>** *сдвиговое-выражение*

реляционное-отношение **<=** *сдвиговое-выражение*

реляционное-отношение **>=** *сдвиговое-выражение*

Ограничения

Должно иметь место одно из следующего:

- оба операнда имеют существующийтип;

- оба операнда являются указателями на квалифицированные или неквалифицированные версии совместимых типов объектов.

Семантика

Каждый из операторов **<** (меньше чем), **>** (больше чем), **<=** (меньше или равно) и **>=** (больше или равно) дает 1, если указанное отношение истинно, и 0, если оно ложно.

Результат сравнения имеет тип **`int`**.

Выражение `a < b < c` не интерпретируется как в обычной математике.

Как указывает синтаксис, оно означает **`(a < b) < c`**; другими словами, «если **a** меньше **b**, выполняется сравнение 1 с **c**, в противном выполняется сравнение 0 с **c**».

Если оба операнда имеют арифметический тип, выполняются обычные арифметические преобразования. Указатель на объект, который не является элементом массива, ведет себя так же, как указатель на первый элемент массива единичной длины с типом объекта в качестве его типа элемента.

Указатели

Когда сравниваются два указателя, результат зависит от относительного расположения в адресном пространстве указанных объектов. Если два указателя на типы объектов оба указывают на один и тот же объект или оба указывают элемент, следующий за последним элементом одного и того же массива, они считаются равными.

Если указанные объекты являются членами одного и того же агрегатного объекта, указатели на элементы структуры, объявленные позже, считаются большими, чем указатели на элементы, объявленные в структуре ранее, а указатели на элементы массива с большими значениями индекса считаются большими, чем указатели на элементы того же массива с меньшими значениями индекса.

Все указатели на члены одного и того же объекта объединения считаются равными.

Оператор равенства

Синтаксис

выражение-равенства:

реляционное-отношение

выражение-равенства **==** *реляционное-отношение*

выражение-равенства **!=** *реляционное-отношение*

Ограничения

Должно иметь место одно из следующего :

- оба операнда имеют арифметический тип;

- оба операнда являются указателями на квалифицированные или неквалифицированные версии совместимых типов;

- один операнд является указателем на тип объекта, а другой — указателем на квалифицированную или неквалифицированную версию **void**;

- один операнд является указателем, а другой — константой нулевого указателя.

Семантика

Операторы == (равно) и != (не равно) аналогичны операторам отношений, за исключением их более низкого приоритета.

Результат имеет тип **int**.

Из-за приоритетов **a < b == c < d** равно 1, когда **a < b** и **c < d** имеют одинаковое значение истинности.

Каждый из операторов выдает 1, если указанное отношение истинно, и 0, если оно ложно.

Для любой пары операндов верно только одно из соотношений равенства.

Если оба операнда имеют арифметический тип, выполняются обычные арифметические преобразования.

Значения комплексных типов равны тогда и только тогда, когда их действительные и мнимые части равны.

Любые два значения арифметического типа из доменов разных типов равны тогда и только тогда, когда результаты их преобразований в (комплексный) тип результата, определяемые обычными арифметическими преобразованиями, равны.

В противном случае, по крайней мере, один операнд является указателем.

Если один операнд является указателем, а другой — константой нулевого указателя, константа нулевого указателя преобразуется в тип указателя.

Если один операнд является указателем на тип объекта, а другой — указателем на квалифицированную или неквалифицированную версию **`void`**, первый преобразуется в тип последнего.

Два указателя считаются равными в том и только в том случае, если оба являются нулевыми указателями, либо оба являются указателями на один и тот же объект (включая указатель на объект и подобъект в его начале) или на функцию, либо оба являются указателями на элемент, следующий после последнего элемента того же массива, либо один — указатель на на элемент, следующий после последнего элемента массива, а другой — указатель на начало другого массива, который непосредственно следует за первым объектом массива в адресном пространстве.

Побитовый оператор И (AND)

Синтаксис

AND-выражение:

выражение-равенства

AND-выражение **&** *выражение-равенства*

Ограничения

Каждый из операндов должен иметь целочисленный тип.

Семантика

Над операндами выполняются обычные арифметические преобразования.

Результатом двоичного оператора **&** является побитовое **И** операндов (то есть каждый бит в результате устанавливается, если и только если установлен каждый из соответствующих битов в преобразованных операндах).

Побитовый оператор исключающего ИЛИ (XOR)

Синтаксис

исключающее-OR-выражение:

AND-выражение

исключающее-OR-выражение **^** *AND-выражение*

Ограничения

Каждый из операндов должен иметь целочисленный тип.

Семантика

Над операндами выполняются обычные арифметические преобразования.

Результатом оператора **^** является побитовое исключающее ИЛИ операндов (то есть каждый бит в результате устанавливается, если и только если установлен ровно один из соответствующих битов в преобразованных операндах).

Побитовый оператор включающего ИЛИ (OR)

Синтаксис

включающее-OR-выражение:

исключающее-OR-выражение

включающее-OR-выражение | *исключающее-OR-выражение*

Ограничения

Каждый из операндов должен иметь целочисленный тип.

Семантика

Над операндами выполняются обычные арифметические преобразования.

Результат оператора | — это побитовое ИЛИ операндов (то есть каждый бит в результате устанавливается тогда и только тогда, когда установлен хотя бы один из соответствующих битов в преобразованных операндах).

Логический оператор И (AND)

Синтаксис

логическое-AND-выражение:

включающее-OR-выражение

логическое-AND-выражение **&&** *включающее-OR-выражение*

Ограничения

Каждый из операндов должен иметь скалярный тип.

Семантика

Оператор **&&** выдает 1, если оба его операнда не равны 0; в противном случае он возвращает 0.

Результат имеет тип **`int`**.

В отличие от побитового двоичного оператора **&**, оператор **&&** выполняет вычисление строго слева направо, т.е. к моменту вычисления второго операнда, первый уже будет вычислен.

Если первый операнд равен 0, второй операнд не вычисляется.

Логический оператор ИЛИ (OR)

Синтаксис

логическое-OR-выражение:

логическое-AND-выражение

логическое-OR-выражение `||` *логическое-AND-выражение*

Ограничения

Каждый из операндов должен иметь скалярный тип.

Семантика

Оператор `||` выдает 1, если любой из его операндов не равен 0; в противном случае он возвращает 0.

Результат имеет тип **`int`**.

В отличие от побитового двоичного оператора `|`, оператор `||` выполняет вычисление строго слева направо, т.е. к моменту вычисления второго операнда, первый уже будет вычислен.

Если первый операнд неравен 0, второй операнд не вычисляется.

Условный оператор

Синтаксис

условное-выражение:

логическое-OR-выражение

логическое-OR-выражение **?** *выражение* **:** *условное-выражение*

Ограничения

Первый операнд должен иметь скалярный тип.

Для второго и третьего операндов должно быть одно из следующих:

- оба операнда имеют арифметический тип;
- оба операнда имеют одинаковую структуру или тип объединения;
- оба операнда имеют тип **`void`**;
- оба операнда являются указателями на квалифицированные или неквалифицированные версии совместимых типов;
- один операнд является указателем, а другой — константой нулевого указателя;
- один операнд является указателем на тип объекта, а другой — указателем на квалифицированную или неквалифицированную версию **`void`**.

Семантика

Вычисления всех трех операндов выполняются в строгой последовательности.

Вычисляется первый операнд;

Второй операнд вычисляется только в том случае, если первый не равен 0.

Третий операнд вычисляется, только в том случае, если первый равен 0.

Результатом является значение второго или третьего операнда, в зависимости от того, что вычислено, преобразованное в тип, как описано ниже.

Условное выражение не дает **lvalue**.

Если и второй, и третий операнды имеют арифметический тип, типом результата является тип, который определяется обычными арифметическими преобразованиями, если бы они применялись к этим двум операндам.

Если оба операнда имеют тип структуры или тип объединения, результат будет иметь этот тип.

Если оба операнда имеют тип **void**, результат имеет тип **void**.

Если и второй, и третий операнды являются указателями, или один из них является константой нулевого указателя, а другой является указателем, типом результата будет указатель на тип, квалифицированный всеми квалификаторами типов, которые присутствуют в обоих операндах.

Кроме того, если оба операнда являются указателями на совместимые типы или на версии с разными квалификациями совместимых типов, тип результата является указателем на версию с определенными квалификаторами составного типа.

Если один операнд является константой нулевого указателя, результат будет иметь тип второго операнда. Если один операнд является указателем на **void** или квалифицированной версией **void**, и в этом случае тип результата представляет собой указатель на соответствующую квалифицированную версию **void**.

Пример

Общий тип результата, если второй и третий операнды являются указателями, определяется в два независимых этапа. Присвоенные квалификаторы не зависят от того, имеют ли два указателя совместимые типы.

Пусть даны объявления:

```
const void   *c_vp;
void         *vp;
const int    *c_ip;
volatile int *v_ip;
int          *ip;
const char   *c_cp;
```

Третий столбец в следующей таблице является общим типом, который будет иметь результат условного выражения, первые два столбца в котором являются вторым и третьим операндами (в любом порядке):

```
c_vp   c_ip    const void *
v_ip   0       volatile int *
c_ip   v_ip    const volatile int *
vp     c_cp    const void *
ip     c_ip    const int *
vp     ip      void *
```

Оператор присваивания

Синтаксис

выражение-присваивания:

условное-выражение

унарное-выражение оператор-присваивания выражение-присваивания

оператор-присваивания: один из

= *= /= %= += -= <<= >>= &= ^= |=

Оператор присваивания в качестве своего левого операнда должен иметь изменяемое **lvalue**.

Семантика

Оператор присваивания сохраняет значение в объекте, обозначенном левым операндом.

Значением выражения присваивания является значение левого операнда после присваивания, но оно не является **lvalue**.

Тип выражения присваивания — это тип, который будет иметь левый операнд после преобразования к **lvalue**.

Обновление значения левого операнда является сторонним эффектом и оно происходит только после вычислений значения левого и правого операнда.

Вычисление операндов не являются последовательными.

Простое присваивание

Семантика

В простом присваивании (**=**) значение правого операнда преобразуется в тип выражения присваивания и заменяет значение, хранящееся в объекте, обозначенном левым операндом.

Если значение, хранящееся в объекте, считывается из другого объекта, который каким-либо образом перекрывается в памяти с первым объектом, то это перекрытие должно быть точным, а два объекта должны иметь квалифицированные или неквалифицированные версии совместимого типа, в противном случае поведение не определено.

Ограничения

Должно выполняться одно из следующего:

- левый операнд имеет атомарный, квалифицированный или неквалифицированный арифметический тип, а правый имеет арифметический тип;

- левый операнд имеет атомарную, квалифицированную или неквалифицированную версию типа структуры или типа объединения, совместимого с типом правого операнда;

- левый операнд имеет атомарный, квалифицированный или неквалифицированный тип указателя, и (учитывая тип, который левый операнд будет иметь после преобразования в **lvalue**) оба операнда являются указателями на квалифицированные или неквалифицированные версии совместимых типов, а тип, на который указывает левый, имеет все классификаторы типа, на который указывает правый;

- левый операнд имеет атомарный, квалифицированный или неквалифицированный тип указателя и (учитывая тип, который левый операнд будет иметь после преобразования в **lvalue**) один операнд является указателем на тип объекта, а другой — указателем на квалифи-

цированную или неквалифицированную версию **`void`**, и тип, на который указывает левый, имеет все квалификаторы типа, на который указывает правый;

- левый операнд является атомарным, квалифицированным или неквалифицированным указателем, а правый — константой нулевого указателя;

- левый операнд имеет тип атомарный, квалифицированный или неквалифицированный **`_Bool`**, а правый — указатель.

Пример 1.
Во фрагменте программы

```
int  f(void);
char c;                 // ???
/* ... */
if ((c = f()) == -1)    // ???
/* ... */
```

значение типа **int**, возвращаемое функцией, может быть усечено при сохранении в **char**, а затем преобразовано обратно до ширины типа **int** перед сравнением.

В реализации, в которой «обычный» **char** имеет тот же диапазон значений, что и **unsigned char** (и **char** меньше, чем **int**), результат преобразования не может быть отрицательным, поэтому операнды сравнения никогда не могут сравниться, как равные.

Поэтому для полной переносимости переменная **c** должна быть объявлена как **int**.

Пример 2.

Во фрагменте программы

```
char c;
int  i;
long l;
l = (c = i);
```

значение **i** преобразуется в тип выражения присваивания **c = i**, то есть тип **char**. Значение выражения, заключенного в скобки, затем преобразуется в тип внешнего выражения присваивания, то есть тип **long int**.

Пример 3.

Рассмотим фрагмент:

```
const char **cpp;
char        *p;
const char   c = 'A';
cpp  = &p;            // нарушение ограничений
*cpp = &c;            // допустимо
*p   =  0;            // допустимо
```

Первое присваивание небезопасно, поскольку оно позволит следующему допустимому коду попытаться изменить значение объекта **const c**.

Составное присваивание

Синтаксис операторов

```
*=    /=   %=   +=   -=   <<=   >>=   &=   ^=   |=
```

Семантика

Составное присваивание в форме **E1 op= E2** эквивалентно выражению простого присваивания **E1 = E1 op (E2)**, за исключением того, что значение **E1** вычисляется только один раз, а в отношении вызова функции с неопределенным порядком выполнения операция составного присваивания — это единое вычисление.

Если **E1** имеет атомарный тип, составное присваивание является операцией *чтения-изменения-записи* (RMW — read-modify-write).

Ограничения

Только для операторов **+=** и **-=**, либо левый операнд должен быть атомарным, квалифицированным или неквалифицированным указателем на полный тип объекта, а правый должен иметь целочисленный тип, либо левый операнд должен иметь атомарный, квалифицированный или неквалифицированный арифметический тип, а правый должен иметь арифметический тип.

Для других операторов левый операнд должен иметь атомарный, квалифицированный или неквалифицированный арифметический тип, и (учитывая тип, который будет иметь левый операнд после преобразования в **lvalue**), каждый операнд должен иметь арифметический тип, совместимый с типами, разрешенными соответствующим двоичным оператором.

Замечания

Если может быть сформирован указатель на атомарный объект, а **E1** и **E2** имеют целочисленный тип, то составное присваивание эквивалентно следующей кодовой последовательности, где **T1** — это тип **E1**, а **T2** — это тип **E2**:

```
T1 *addr = &E1;
T2  val  = (E2);
T1  old  = *addr;
T1  new;
do {
    new = old op val;
} while (!atomic_compare_exchange_strong(addr, &old, new));
```

где **new** является результатом операции.

Оператор запятая

Синтаксис

выражение:

выражение-присваивания

выражение **,** *выражение-присваивания*

Семантика

Левый операнд оператора запятая вычисляется как выражение, имеющее тип **`void`** (не имеющее типа); между его вычислением и вычислением правого операнда существует *точка упорядочения*.

Затем вычисляется правый операнд. Конечный результат имеет этот тип и значение.

Оператор запятая не дает **`lvalue`**.

Оператор запятая (как описано в этом подпункте) не может появляться в контекстах, где запятая используется для разделения элементов в списке (таких как аргументы функций или списки инициализаторов). Однако, в таких контекстах он может использоваться в выражении, взятом в скобки, или во втором выражении условного оператора. **Пример**

В вызове функции

```
f(a, (t=3, t+2), c)
```

функция имеет три аргумента, второй из которых имеет значение 5.

Константное выражение

Синтаксис

константное-выражение:
    условное-выражение

Описание

Константное выражение вычисляется во время трансляции, а не во время выполнения, и, соответственно, может использоваться в любом месте, где может быть константа.

Ограничения

Константные выражения не должны содержать операторов присваивания, инкремента, декремента, вызова функции или оператора запятой, за исключением случаев, когда они содержатся в подвыражении, которое не вычисляется. Например, обычно не вычисляется операнд оператора **`sizeof`** или **`_Alignof`**.

Каждое константное выражение должно вычисляться как константа, которая находится в диапазоне представимых значений для своего типа.

Семантика

В некоторых контекстах требуется выражение, которое вычисляет константу. Если выражение с плавающей точкой вычисляется в среде трансляции, арифметический диапазон и точность должны быть как минимум такими же большими, как если бы выражение вычислялось в среде исполнения.

Целочисленные константные выражения требуется в ряде контекстов, таких как размер элемента битового поля структуры, значение константы перечисления и размер массива неизменяемой длины.

Целочисленные константные выражения должны иметь целочисленный тип и должны иметь только такие операнды, которые являются целочисленными константами, константами перечисления, символьными константами, выражениями **`sizeof`**, результатами которых будут целочисленные константные выражения, **`_Alignof`** и константы с плавающей точкой, которые являются непосредственными операндами операторов приведения типа.

Операторы приведения типа в целочисленных константных выражениях должны преобразовывать только арифметические типы в целочисленные типы, исключая части операндов операторов **`sizeof`** или **`_Alignof`**.

Константное выражение в инициализаторах должно быть или вычислять одно из следующего:

- выражение арифметической константы;
- константу нулевого указателя;
- адресную константу;
- адресную константу для завершенного типа объекта, плюс-минус целочисленное константное выражение.

Арифметическое константное выражение должно иметь арифметический тип и иметь только такие операнды, которые являются целочисленными константами, плавающими константами, константами перечисления, символьными константами, выражениями **`sizeof`**, результаты которых являются целочисленными константами, и выражениями **`_Alignof`**.

Операторы приведения в арифметических константных выражениях должны преобразовывать только арифметические типы в арифметические типы, за исключением частей операндов операторов **`sizeof`** или **`_Alignof`**.

Адресная константа — это нулевой указатель, указатель на **lvalue**, обозначающее объект статической длительности хранения, или указатель на имя функции.

Он должен быть создан явно с использованием унарного оператора **&** или целочисленной константы, приведенной к типу указателя, или неявно с помощью выражения с типом массива или типом функции.

В процессе создания адресной константы могут использоваться оператор индекса массива **[]**, оператор доступа к члену **.** и оператор **->** , операторы получения адреса **&** и разыменования *****, приведение типа указателя, но с помощью этих операторов не должно быть доступно значение объекта.

Семантические правила для оценки константного выражения такие же, как и для непостоянных выражений. Таким образом, в следующей инициализации,

```
static int i = 2 || 1 / 0;
```

выражение является допустимым целочисленным константным выражением со значением единица.