GeekBrains

Основы C++

Операции и выражения

На этом уроке

1. Узнаем в чём разница между ссылкой и указателем
2. Научимся арифметике указателей
3. Рассмотрим все операции, доступные программисту на языке C++
4. Подробно изучим битовые операции

Оглавление

Операции и выражения

Как мы уже говорили ранее, совокупность операндов и операций в языке Си образуют выражение.

Операции	Выражения
- Присваивания - Арифметические - Сравнения - Логические - Побитовые - Специальные	- Имеют тип и значение - Совокупность операндов и операций - Синтаксически строги

При составлении выражения необходимо соблюдать синтаксические правила. Каждое выражение, как и операнд, имеет тип и значение. Тип операндов может неявным образом влиять на результат.

Операции

В языке C++ существуют четыре категории операций:

- *ссылки,*
- *унарные операции,*
- *бинарные*
- *тернарные операции.*

Операции ссылки используются при доступе к элементам массивов и структур. Унарные операции воздействуют на одно значение или выражение. В бинарных операциях участвуют два выражения, а в тернарных - три. В особый класс бинарных операций можно выделить операции присваивания. Как видно из таблицы, приведенной выше, в дополнение к традиционным арифметическим и логическим операциям, операциям сравнения и присваивания, в языке Си предусмотрены побитовые операции и

специальные операции, такие, как операция "запятая", операции над указателями и сокращенные версии большинства "традиционных" операций.

Операции присваивания

Синтаксис оператора присваивания следующий:

```
lvalue = выражение;
```

Например: `int x = 5;`

Мы уже говорили об этой операции, когда говорили об инициализации переменных в языке C++, теперь дадим некоторые уточнения. Значением оператора присваивания служит значение выражения, которое присваивается объекту lvalue (в этом контексте lvalue можно трактовать как "left value", в смысле, "значение слева от оператора присваивания"). Операторы присваивания могут быть вложенными и составными, например:

```
max = min = 0;
sum = sum + count;
a += 2;
```

Операции присваивания (=) группируются (исполняются) справа налево. Поэтому в первом примере вначале будет выполнено присваивание `min = 0` в результате которого значение `min` станет равным нулю. Затем это же значение будет присвоено переменной `max`. Во втором примере значением выражения `sum` будет значение правой части оператора присваивания, т.е. (`sum + count`). Третий пример является сокращенным вариантом записи `a = a + 2;` Вот еще примеры сокращенной записи:

```
offset /= 2; эквивалентно offset = offset / 2;
array[i + 2] -= 10;  эквивалентно array[i + 2] = array[i + 2] - 10;
```

Применение сокращенных операций присваивания дает ряд преимуществ. Во-первых, их запись короче. Во-вторых, компилятором они могут быть реализованы эффективнее, поскольку объект `lvalue` входит в состав выражения только один раз. В последнем из приведенных выше примеров вычисление индекса массива должно быть выполнено в правом выражении два раза, а в левом - только один раз.

В языке Си использование вложенных и составных операторов присваивания широко распространено и представляет особый интерес в связи с приоритетами операций. В частности, приоритет операции присваивания ниже приоритетов операций сравнения, логических операций или тернарной операции, она находится почти в самом конце списка операций:

Класс операций	Операции	Ассоциативность
первичные	`()` `[]` `->` `.`	слева направо
унарные	`!` -(унарный) `*`(разыменование) `&`(вычисление адреса) `++` `--`	справа налево

	`sizeof`	
мультипликативные	`/` `*` `%`	слева направо
аддитивные	`+` `-`	слева направо
побитовый сдвиг	`<<` `>>`	слева направо
отношения	`<` `<=` `>` `>=`	слева направо
проверка равенства	`==` `!=`	слева направо
побитное И	`&`	слева направо
побитное исключающее ИЛИ	`^`	слева направо
побитное ИЛИ	`|`	слева направо
логическое И	`&&`	слева направо
логическое ИЛИ	`||`	слева направо
условная	`?:`	справа налево
присваивания	`=` `+=` `-=` `*=` и т.д.	справа налево
запятая	`,`	слева направо

Строки таблицы показаны в порядке уменьшения приоритетов от верхних строк к нижним. Поэтому, например, в выражении:

```
if ((sum = sum + count) < max) ...;
```

необходимо указывать скобки, если желательно, чтобы значение оператора присваивания сравнивалось со значением переменной `max`. Если в этом выражении опустить скобки: `sum = sum + count < max;` то ввиду соотношений приоритетов вначале выполнится вычисление выражения `sum + count`, затем оно будет сравнено со значением переменной `max`. Полученный результат сравнения (`0` или `1`), а вовсе не `sum + count` будет присвоен переменной `sum`. При равном приоритете унарные операции группируются справа налево, а бинарные - слева направо.

Порядок вычисления коммутативных операций в языке Си не регламентируется. Например, порядок вызовов функции при вычислении выражения. Кроме того, в языке Си не определен порядок вычисления аргументов функции при ее вызове. Неправильный учет приоритетов является одной из распространенных ошибок при программировании на языке Си, поэтому постарайтесь хорошо запомнить соотношения между приоритетами операций или, если не уверены, пользоваться круглыми скобками для выставления приоритетов.

Операция присваивания не может быть применена ко всему массиву. Однако в состав подавляющего большинства стандартных библиотек входят функции, которые копируют содержимое массивов, по крайней мере, массивов символов. Эти функции будут обсуждаться на следующих занятиях.

Многие современные компиляторы языка Си позволяют выполнять присваивания структур одного и того же типа с помощью операции `=`. Это означает, что код присваивания структуры:

```
struct STOCK current, old;
old = current;
```

копирует все содержимое структуры `current` в структуру `old`. Чтобы выяснить, обеспечивает ли Ваш компилятор эту возможность, обратитесь к его документации.

Арифметические операции

Эти операции задают обычные действия над операндами любого арифметического типа. В языке Си существует одна унарная и пять бинарных арифметических операций, как показано ниже:

Операция	Значение	Пример	Тип
`-`	изменение знака	`-a`	унарный
`*`	умножение	`a * b`	бинарный
`/`	деление	`a / b`	бинарный
`%`	остаток от деления	`a % b`	бинарный
`+`	сложение	`a + b`	бинарный
`-`	вычитание	`a - b`	бинарный

Не будем подробно останавливаться на традиционных арифметических бинарных операциях. Но одна из указанных операций, а именно `%`, может оказаться незнакомой, или, по крайней мере, непривычной. Это **операция взятия модуля**, т.е. вычисления остатка от деления одного целого числа на другое. Операция взятия модуля выполняется только над целочисленными операндами (для значений типа `float` или `double` она недопустима). Это один из примеров того, как тип операндов влияет на результат.

При выполнении арифметических операций в языке Си автоматически выполняется преобразование типов так, как описывается далее, в разделе “операторы преобразования типов”. При вычислении выражений, в которые входят арифметические операции, язык Си следует правилам приоритета, показанным ранее, при обсуждении операций присваивания. Это не противоречит и обычным правилам арифметики: приоритет умножения и деления равен и он выше, чем приоритет сложения и вычитания, которые между собой равноправны. Для изменения порядка выполнения операций,

предписанного их приоритетами, могут быть использованы скобки, которые тоже являются операцией, а их приоритет - наивысший.

Унарные арифметические операции

В таблице арифметических операций показана только одна унарная операция – смена знака. Как следует из ее названия, применима она к переменным знаковых типов `int`, `short`, `long`, `float`, `double`, `signed char`, и не может быть применена к объектам типа `unsigned`.

```
int val = 5;
val = -val;
```

Кроме изменения знака, в Си определены операции унарное увеличение (`++`) и унарное уменьшение (`--`). Их ещё называют инкремент и декремент, соответственно. Эти операции не выделены отдельной строкой в вышеприведенной таблице, поскольку являются сокращенной формой записи операций сложения и вычитания, но операндом для них служит только адресуемое значение. Эти операции добавляют или вычитают единицу из значения того объекта, на который указывает адресуемое значение, и присваивают ему полученный результат. Таким образом, они выполняют неявное присваивание.

В таблице приоритетов унарные операции находятся на втором сверху уровне. При равном приоритете унарные операции группируются справа налево, а бинарные арифметические - слева направо. Знак унарной операции может стоять как перед, так и после операнда (`++x;` `x++;`) или (`--x;` `x--;`). Первая запись в каждой из скобок называется префиксной, вторая - постфиксной. Обе записи транслятор интерпретирует соответственно как:

```
x = x + 1; // (++x; x++;)
x = x - 1; // (--x; x--;)
```

однако, исполняться префиксная и постфиксная запись будут в разной последовательности. Вот пример префиксного увеличения индекса массива при доступе к следующему элементу массива:

```
a  =  array[++x];
// или то же самое, но записанное в традиционном стиле:
x = x + 1;
a = array[x];
```

Постфиксные операции увеличения и уменьшения осуществляются почти так же. Значение адресуемого объекта по-прежнему увеличивается (или уменьшается) на единицу, и только затем этот объект модифицируется. При этом в выражении используется старое значение объекта, которое он имел до модификации. Вот пример постфиксного увеличения:

```
a = array[x++];
//традиционном форма этого выражения:
a = array[x];
x = x + 1;
```

Обратите внимание на отличие от префиксной записи. При постфиксном увеличении в качестве значения индекса массива берется то значение переменной `x`, которое она имела до операции увеличения. Затем этой переменной присваивается новое значение.

При использовании префиксных и постфиксных операций увеличения или уменьшения тип результата тот же, что и тип адресуемого значения, над которым выполняется операция. Однако при использовании операций унарного увеличения и унарного уменьшения, изменяющих значение операнда, в одном выражении с бинарными операторами над этим операндом, возможен побочный эффект, зависящий от конкретной реализации транслятора. Так, в следующем примере:

```
int y, x = 2;
y = x++ - ++x;
```

не определено, какие значения будут использованы в операторе вычитания. Приведённый пример является классическим “запутывающим” вопросом на собеседованиях, поскольку однозначного ответа на него не может существовать.

Операции сравнения

Операции сравнения (известные также под именем "операции отношения") используются для проверки условий. Результатом выполнения может быть `true` (истина), если выражение удовлетворяет условию, или `false` (ложь), если не удовлетворяет условию. При этом в языке C результат выполнения представляется значением типа `int`, где `true` эквивалентно значению `1`, а `false` - значению `0`. Но в более поздней редакции (C99 `#include <stdbool.h>`) и языке C++ появился специальный булев тип, имеющий литеральные значения. Что никак не мешает им всё равно компилироваться в `0` и `1`.

Операция	Значение	Пример
==	равно	a == b
!=	не равно	b != limit
<	меньше	a < b
<=	меньше или равно	(a - 10) <= limit
>	больше	b > a
>=	больше или равно	(a + 10) >= limit

Приоритет операций сравнения меньше, чем у бинарных арифметических операций. Приоритет операций `<`, `>`, `<=`, `>=` одинаков и выше, чем у операций `==` и `!=`. Таким образом, в сравнении

```
  count < sizeof(iarray) / 2;
```

скобки можно не указывать, поскольку это выражение эквивалентно выражению

```
count < (sizeof(iarray) / 2);
```

однако в случае сомнения и для облегчения чтения программы хорошим тоном является использовать скобки, чтобы явно задать порядок выполнения таких операций.

В выражениях операции отношения группируются слева направо. Напоминаем, что результатом выражения, включающего в себя операции отношения, может быть или ненулевое значение ("истина"), или нулевое значение целого типа ("ложь"). Таким образом, после выполнения `int x = count < sizeof(iarray) / 2;` переменной `int x` будет присвоено значение либо `1`, либо `0`.

Обратите внимание: одна и та же операция сравнения может дать разный результат в зависимости от того, являются ли операнды знаковыми, или беззнаковыми (`unsigned`). Это еще один пример того, как тип операндов влияет на результат операции. С помощью операций отношения можно сравнивать и такие объекты, как указатели, о чем поговорим далее.

Логические операции

Логические операции (известные также под именем "операции связки", чтобы не путать с побитовыми логическими операциями, о которых поговорим чуть позже) служат для связывания выражений, включающих в себя операции отношения.

Операция	Значение	Пример
`&&`	And (И)	`(a < b) && (b < c)`
`||`	Or (ИЛИ)	`(a > b) || (b > c)`
`!`	Not (НЕ)	`!(a == b)`

Результаты применения логических операций определены их таблицами истинности:

a	b	&&
0	0	0
0	1	0
1	0	0
1	1	1

a	b	||
0	0	0
0	1	1
1	0	1
1	1	1

a	!
0	1
1	0

И возвращает истину только если оба операнда истинны, ИЛИ возвращает истину если хотя бы один операнд истинный, НЕ возвращает истину когда операнд ложный. Помните, что ненулевое целое значение представляет логическое значение "истина", а нулевое - "ложь". Поэтому операция логического отрицания изменяет ненулевое значение на нуль, а нуль - на единицу.

Приоритет логических операций `&&` и `||` ниже приоритета операций `==` и `!=`, определяющих равенство или неравенство значений. Выражения, соединенные операциями `&&` и `||`, также вычисляются слева направо, и при этом вычисление выражения **прекращается, как только его результат становится**

однозначно определен (обратите внимание! Игнорирование этого факта часто приводит к ошибкам, особенно при сравнении результатов работы вызываемыхфункций). Таким образом, при выполнении оператора:

```
if (count < sizeof(iarray) / sizeof(int) && iarray[count] > 0) ...
```

сначала будет вычислено отношение `count < sizeof(iarray) / sizeof(int)`. Если его значение "ложь", то заранее можно сказать, что и значение всего выражения тоже "ложь". На этом вычисление выражения **будет прекращено** поскольку при любом значении второго выражения результат всё равно останется ложным. Но если значение данного отношения "истина", то будет вычислено также и отношение `iarray[count] > 0`, а после этого к двум полученным значениям будет применена операция `&&`. Логические бинарные операции группируются слева направо.

Побитовые операции

Как следует из их названия, побитовые (термин "побитные" также справедлив) операции действуют на каждый бит соответствующего операнда. Вот перечень этих операций в языке C++ с примерами:

Операция	Значение	Пример
`~`	побитное отрицание	`~result`
`&`	побитное И	`value & 0377`
`|`	побитное ИЛИ	`result | 040`
`^`	побитное исключающее ИЛИ	`value ^ -1`
`<<`	сдвиг влево	`value << 2`
`>>`	сдвиг вправо	`value >> 2`

Операция побитного отрицания (известная также под именем операции дополнения до единицы) является унарной, остальные операции - бинарные. (Не путать операцию `~` побитного отрицания с операцией `!`). Операция побитного отрицания изменяет значения всех битов целого числа на противоположные. Каждый единичный бит становится нулевым, а каждый нулевой бит единичным. Приоритет этой операции тот же, что и у других унарных операций, и группируется она также справа налево.

Побитовые операции применимы к объектам целого и символьного типов (типы `int`, `short`, `long`, `char`, `signed`, `unsigned`, а также вариациям этих типов). Побитовые операции не могут быть применены к объектам типа `float` или `double`, в связи с особенностями хранения значений этих типов данных

Для понимания этих операций надо прежде всего выяснить, как они действуют на отдельные биты. Если у нас есть два операнда, каждый из которых может принимать значение 0 или 1, то для каждой операции существует четыре комбинации значений операндов. К таблицами истинности логических

операторов (с учётом замены `!` на `~`, `&&` на `&` и `||` на `|`) добавляется ещё одна таблица для побитного исключающего ИЛИ (возвращает истину, если операнда различаются):

a	b	^
0	0	0
0	1	1
1	0	1
1	1	0

Побитовые операции выполняют действия над каждыми взаимно соответствующими битами двух операндов. Следующие записи показывают результат действия операций ИЛИ и Исключающее ИЛИ над 16-битовыми операндами:

ИЛИ	Исключающее ИЛИ
0001001101100011 (4963) 0100001100100001 (17185) ---------------- 0101001101100011 (21347)	0001001101100011 (4963) 0100001100100001 (17185) ---------------- 0101000001000010 (20546)

Каждая из побитовых операций служит определенным целям. Операция `&` (И) полезна для проверки того, что конкретные биты установлены (т.е. равны единице), а также для обнуления (маскирования) битов. Она полагает бит результата равным единице в том, и только том случае, если оба соответствующих бита операндов равны `1`.

Операция `|` (ИЛИ) полезна для установки битов. Бит результата полагается равным единице, если хотя бы один из соответствующих битов операндов равен единице.

Операция `^` (Исключающее ИЛИ) полезна для проверки несовпадения битов (бит результата полагается равным `1` только в том случае, если соответствующие биты операндов различны). Она может быть использована также для "переключения" битов из `0` в `1` и обратно.

Если два операнда коммутативных бинарных операций `&`, `|` или `^` являются выражениями, то порядок вычисления этих выражений в языке C++ не регламентируется. После вычисления операндов эти операции группируются слева направо.

Побитовые операции сдвига хорошо известны тем, кто хотя бы немного программировал на языке ассемблера. Там типов операций сдвига гораздо больше: арифметические, логические и т.п. Эти операции бинарные. Операция сдвига влево обозначается символом `<<`. Результатом вычисления бинарного выражения

```
a << n;
```

является совокупность битов, полученная из значения выражения `a`, указанного в левой части, путем сдвига влево на число битов `n`, задаваемое выражением, указанным в правой части. Освободившиеся справа биты заполняются нулями. Попутно заметим, что сдвиг битов влево на одну позицию эквивалентен умножению операнда на два. То есть операция сдвига влево на `n` позиций это *умножение операнда на двойку в степени* `n`. Если указан сдвиг на отрицательное число битов, то в языке Си результат выполнения такого выражения считается неопределенным.

Побитовый сдвиг вправо обозначается символом `>>`. Результатом вычисления бинарного выражения

```
b >> m;
```

является совокупность битов, полученная из значения выражения `b`, указанного в левой части, путем сдвига вправо на число битов `m`, указанном в правой части. Принципиальное отличие от сдвига влево состоит в том, что способ заполнения освобождающихся слева битов зависит от типа выражения, указанного в левой части. Если левый операнд не имеет знака, то освобождающиеся биты заполняются нулями. Однако если левый операнд является целым значением со знаком, то в результате правого сдвига происходит "расширение знака", т.е. заполнение освобождающихся битов левого операнда значением знакового бита (`1` для отрицательного значения левого операнда, `0` - для положительного).

Расширение знака демонстрируется следующей программой:

```
main() {
 int x = 0x8000; // Установить знаковый бит
 unsigned int z = 0x8000; // Установить старший бит
 // Выражение       Двоичное значение       Шестнадцатеричное значение
 // x;              // 1000000000000000     0x8000
 x >>= 1;          // 1100000000000000*    0xC000
 x >>= 2;          // 1110000000000000*    0xE000
 // * - обратите внимание на расширение знака
 // обратите внимание на использование операции сдвига совместно с операцией
присваивания =>>
 // аналогично для z
 // unsigned z;    // 1000000000000000     0x8000
 z >>= 1;          // 0100000000000000*    0x4000
 z >>= 2;          // 0010000000000000*    0x2000
}
```

Приоритет операций сдвига меньше приоритета операций `+` и `-`, а группируются они слева направо.

Специальные операции

Операция скобки

Операции `()` мы касались, когда говорили об операциях присваивания. “Круглые скобки” там использовались для изменения последовательности операций внутри выражения. В этом контексте

“круглые скобки” могут быть вложенными `((((...))))` в соответствии с обычными правилами арифметики. Также `()` используются как элемент синтаксиса выражений и функций языка С, например при явном преобразовании типов переменных, при передаче параметров в функции и т.п.

“Квадратные скобки” `[]` используются для инициализации размера массива и для указания индексов массивов, о чем мы тоже говорили на соответствующем занятии.

“Фигурные скобки” `{}` используются для инициализации массивов, при объявлении и инициализации структур, а также для выделения блоков операторов и тел функций, о чем поговорим чуть позже.

Операция точка

Ссылка на элемент структуры осуществляется с помощью операции, обозначаемой точкой. Выражение `astruct.member` представляет значение элемента `member` структуры `astruct`. Так как структуры могут быть вложенными, то это справедливо и для операции ссылки на элемент структуры. Например `astruct.bstruct.member` представляет элемент member структуры `bstruct`, которая, в свою очередь, является элементом структуры `astruct`. Точка находится на самом верху таблицы приоритета операторов.

Операция sizeof

Операция `sizeof` выполняется на этапе компиляции программы и дает константу, которая равна числу байтов, требуемых для хранения в памяти данного объекта. Объектом может быть имя переменной, массива, структуры или просто спецификация типа. Применение этой операции демонстрировалось нами ранее. Приведем еще один пример использования операции `sizeof`:

```
int count, iarray[10];
for (count = 0; count < sizeof(iarray) / sizeof(int); count++)
    printf("iarray[%d] = %d\n", count, iarray[count]);
```

Заметьте, что результатом операции `sizeof(iarray)` будет 20, если для хранения целого значения выделяется два байта. Таким образом, число элементов массива равно размеру памяти, необходимой для хранения массива, делённому на размер памяти, необходимой для хранения одного его элемента.

Обратите также внимание на использование унарной операции ++ для увеличения значения счетчика. Это типичный стиль программирования на языке С.

Применение операции `sizeof` всюду, где это возможно, считается хорошим стилем кодирования программы. Если в приведенном примере размер массива `iarray` будет изменен, то после компиляции программы условие в цикле `for` останется правильным (конечно, другая альтернатива - определить именованную константу с помощью директивы `#define` и использовать ее в определении массива и в условии цикла).

Операция преобразования типа

Мы уже затрагивали вопрос преобразования типов, когда говорили о типах переменных. Теперь подробнее. При вычислении выражений компилятор производит, при необходимости, преобразование типов переменных. В языке Си существует три типа преобразований:

- автоматическое дополнение;
- необходимые преобразования при вычислении выражений;
- явное изменения типа выражения по запросу программиста.

Первые два типа преобразования не представляют из себя ничего сложного и делаются автоматически. Вот краткое резюме по этим типам: автоматическое дополнение преобразует типы данных `char` и `short int` в `int`, если в одном выражении встречаются эти типы данных одновременно. Необходимые преобразования были “узаконены” в ANSI C, они производятся в следующей последовательности:

- если один из операндов имеет тип `long double`, другой операнд и результат вычисления преобразуется также к типу `long double`;
- если один из операндов имеет тип `double`, другой операнд и результат вычисления преобразуется к типу `double`;
- если один из операндов имеет тип `float`, другой операнд и результат вычисления преобразуется к типу `float`;
- если операнды имеют тип `char` или `short int`, они преобразуется к типу `int` (автоматическое дополнение);
- если один из операндов имеет тип `long int`, другой операнд и результат вычисления преобразуется к типу `long int`;
- если при вычислении выражения достигнут данный пункт, то оба операнда имеют тип `int`, и результат также будет иметь тип `int`.

Для **явного преобразования типа** программист должен воспользоваться операцией замены типа, синтаксис применения которой следующий:

```
(спецификация_типа) выражение;
```

Значение приведенного выражения будет преобразовано в значение типа `спецификация_типа`. `Спецификация_типа` может быть любым служебным словом, задающим спецификацию типа, например `int`, `short`, `long`, `float` и т.д., и может быть также именем структуры или объединения либо типом указателя (последний вид объектов обсуждаются далее, в теме указателей).

```
int x = 3;
float y;
y = (float) x / 2;  // Результат вычисления y = 1.5;
/* без такого преобразования: */
int x = 3;
float y;
```

```
y = x / 2;  // Результат вычисления y = 1;
```

Ранее мы также рассматривали **макрос typedef**, когда говорили о переопределении типов переменных.
Программист может назначить дополнительное имя любому из типов при помощи инструкции:

```
typedef старый_тип новый_тип;
```

Преобразования типа могут выполняться как по возрастанию, так и по убыванию типов. Результат преобразования от меньшего типа к большему приводит к тому же самому значению, которое, возможно, будет представлено с большей точностью. Но важно обратить внимание, что почти все компиляторы интерпретируют старший бит как знак и могут выполнять расширение знака, копируя знаковый бит в старший байт переменной нового типа. Некоторые могут не выполнять этих действий.

Если заменяется больший тип на меньший, то компилятор отбрасывает избыточные данные. При преобразовании значений с плавающей точкой двойной точности в значения с плавающей точкой одинарной точности выполняется такое округление исходного значения, при котором результат помещается в объекте одинарной точности. При преобразовании значений с плавающей точкой в целые значения отбрасывается (важно, что не округляется, а именно отбрасывается) дробная часть, то есть значение `10.9999` превратится в `10`, а не в `11`. Большие целые значения преобразуются в меньшие целые путем отбрасывания избыточных старших битов.

Тернарная условная операция ?:

Эта операция – единственная в языке Си, которая имеет 3 операнда. Тернарная операция используется для конструирования условных выражений, её также называют оператором выбора. Ее синтаксис следующий:

```
выражение_1 ? выражение_2 : выражение_3
```

Условным выражением называется такое выражение, которое может иметь одно из двух возможных значений. Оно интерпретируется следующим образом: вначале вычисляется `выражение_1`. Если его значение отлично от нуля ("истина"), то вычисляется `выражение_2` (следующее за знаком `?`) и его значение принимается за значение условного выражения. Если же `выражение_1` имеет нулевое значение ("ложь"), то за значение условного выражения принимается значение `выражение_3`.

В следующих фрагментах кодов показаны два способа присваивания переменной max максимального из двух значений:

```
max = (current > next) ? current : next; // Действие тернарной операции
// То же действие, но записанное традиционным способом:
if (current > next) max = current; // Об операторе ветвления позже
else  max = next;
```

Приоритет условной операции ниже приоритета логических операций. Эта операция группируется справа налево.

Операция запятая

Операция "запятая" может быть использована для разделения нескольких выражений. Эта операция чаще всего применяется в операторе for для того, чтобы выполнялось более одного выражения в управляющей части этого оператора. Например, следующий оператор просуммирует первые 10 ненулевых целых значений:

```
for (c = 1, sum = 1; c < 10; c++, sum += c);
```

О самом операторе `for` поговорим позднее, запятая же позволяет здесь поместить несколько выражений вместо одного, обусловленного синтаксисом. В приведенном примере на каждом проходе цикла будет выполнен как оператор `c++`, так и оператор `sum += c`. Также оператор «запятая» используется для связки нескольких выражений. Значение выражения, находящегося с правой стороны, станет значением разделенного запятыми выражения. Например:

```
x = (y = 3, y + 1);
```

Сначала присваивается `3` переменной `y`, а затем `4`. переменной `x`. Скобки необходимы, поскольку оператор «запятая» имеет более низкий приоритет по сравнению с оператором присваивания. Оператор «запятая» вызывает выполнение последовательности действий. Ниже приведен еще один пример:

```
y = 10;
x = (y = y - 5, 25 / y);
```

После выполнения `x` получит значение `5`, поскольку исходным значением `y` было `10`, а затем оно уменьшилось на `5`. Затем `25` поделили на полученное `5` и получили результат. Об операторе «запятая» следует думать как об обычном слове «и» в нормальном русском языке, когда оно используется в выражении «сделай это, и это, и это». Приоритет запятой среди всех операций самый низкий. Эта операция группируется слева направо и используется довольно редко, поскольку довольно сильно снижает читаемость кода.

Операции ссылки

Мы уже говорили о двух из трех операций ссылки, определенных в языке C, а именно операции ссылки на элемент массива (`array[3]`) и элемент структуры (`astruct.bmember`). Здесь напомним: операциям ссылки присвоен очень высокий приоритет и они группируются слева направо.

Ссылка на элемент массива

Операция предназначена для выделения конкретного элемента массива. Чтобы использовать эту операцию, выражение (называемое индексом и обязательно имеющее целое значение) заключают в квадратные скобки и указывают при имени массива, например:

```
array[целочисленное_выражение]
```

Алгоритм операции ссылки на элемент массива следующий:

1. Вычислить выражение, которое служит индексом. Предположим, что результат равен S.
2. Преобразовать S в сдвиг в массиве путем умножения S на размер отдельного элемента массива:
   `сдвиг = S * sizeof(тип_массива);`
3. Вычислить адрес элемента массива по формуле:
   `адрес = имя_массива + сдвиг;`
4. Извлечь значение, находящееся по этому адресу.

Предположим, к примеру, что массив array имеет тип `int`. Тогда ссылка `array[3]` (четвертый элемент) будет найдена как содержимое адреса (`array + (3 * sizeof(int))`. Еще раз напомним, массив **размером n** начинается с нулевого элемента `array[0]`, и заканчивается `n-1` элементом `array[n-1]`.

При работе с многомерными массивами для каждой размерности необходима своя операция ссылки. Выражение `array[i][j]` соответствует элементу двумерного массива array, находящемуся на пересечении строки `i` и столбца `i`. Для массива с размерами `N*M` это означает следующий порядок расположения элементов:

```
two_array[0][0] Первый элемент
two_array[0][1] Второй элемент
two_array[0][2] Третий элемент
....
two_array[N-1][M-1] Последний элемент
```

В действительности операция ссылки на элемент массива может быть использована в **любом** выражении в языке С, результатом вычисления которого является адрес.

Ссылка на элемент структуры

Ссылка на элемент структуры осуществляется с помощью операции, обозначаемой точкой. Выражение `astruct.bmember` представляет значение элемента `bmember` структуры `astruct`. Так как структуры могут быть вложенными, то это справедливо и для операции ссылки на элемент структуры. Например `astruct.bstruct.cmember` представляет элемент `cmember` структуры `bstruct`, которая, в свою очередь, является элементом структуры `astruct`.

Ссылка на элемент структуры с помощью указателя

Операция обозначается как `->` и требуется в том случае, если для ссылки на элемент структуры используется указатель. При обсуждении указателей поговорим об этой операции подробнее.

Операция вычисления (взятия) адреса

```
&операнд; // операндом может быть только lvalue, например, имя переменной
```

Как известно слушателям из предыдущих материалов, для извлечения операндов из памяти компьютера и записи в память, при необходимости, результатов вычислений, надо знать физический адрес в памяти, по которому хранятся эти данные. Распределением адресов памяти в языках

высокого уровня занимается, компилятор совместно с операционной системой компьютера, но дело в том, что язык C/C++ изначально и предназначался, в том числе, и для разработки операционных систем и компиляторов. Поэтому, являясь в полной мере языком высокого уровня абстрагирования от аппаратных платформ и структуры данных, C/C++ сохраняет такую важную возможность ассемблера, как непосредственная работа с адресами памяти, которая затем абстрагирована в нем в понятия индексации, ссылок и работы с указателями.

Про индексацию и ссылки мы поговорили, но очевидно, что для полноценной работы с адресами просто необходимо иметь возможность получить адрес, по которому в памяти расположена переменная или какой-либо другой объект. Один из способов получения адреса объекта состоит в применении унарной операции взятия адреса к переменной и присваивании ее результата переменной-указателю

```
int x, *px; // x - целое значение, px - указатель на целое значение
x = 2; // физический адрес переменной назначили компилятор и ОС
px= &x; // px теперь содержит адрес переменной x мы можем им манипулировать
```

Выражение `&x` означает "адрес объекта `x`". Таким образом, переменной `px` будет присвоено значение указателя на `x` (т.е. адрес объекта `x`). Так как переменная `x` является объектом целого типа, а переменная `px` определена как указатель на объекты целого типа, то показанный выше код допустим. Единственное ограничение применения унарной операции `&` состоит в том, что ее нельзя применять к переменной с классом хранения `register`, а также к полям битов.

Другой способ занесения адреса в указатель состоит в присваивании указателю значения известной константы. Такая необходимость возникает в приложениях, где заранее известны абсолютные адреса, задающие, например, ячейки с описанием состояния аппаратных средств. К переменному указателю, как только ему присвоен допустимый адрес, Вы можете применить унарную операцию взятия косвенного адреса, обозначаемую через `*`.

Операция косвенной адресации (разыменование указателя).

```
*операнд;
```

В данном контексте унарная операция `*` означает уже не умножение, а извлечение значения, находящегося по адресу, на который указывает операнд. Выражение `*px` означает "извлечь значение из области памяти, на которую указывает содержимое переменной `px`". А так как переменная `px` благодаря предшествующему присваиванию (`px = &x;` см. выше пример операции **вычисления (взятия) адреса)** содержит адрес переменной `x`, то выражение `*px` соответствует извлечению значения `x`. Этот путь получения содержимого переменной называется **косвенной адресацией** или **разыменованием**.

Унарная операция косвенной адресации может быть использована также в левой части оператора присваивания для того, чтобы какие-то данные были запомнены в области памяти, на которую

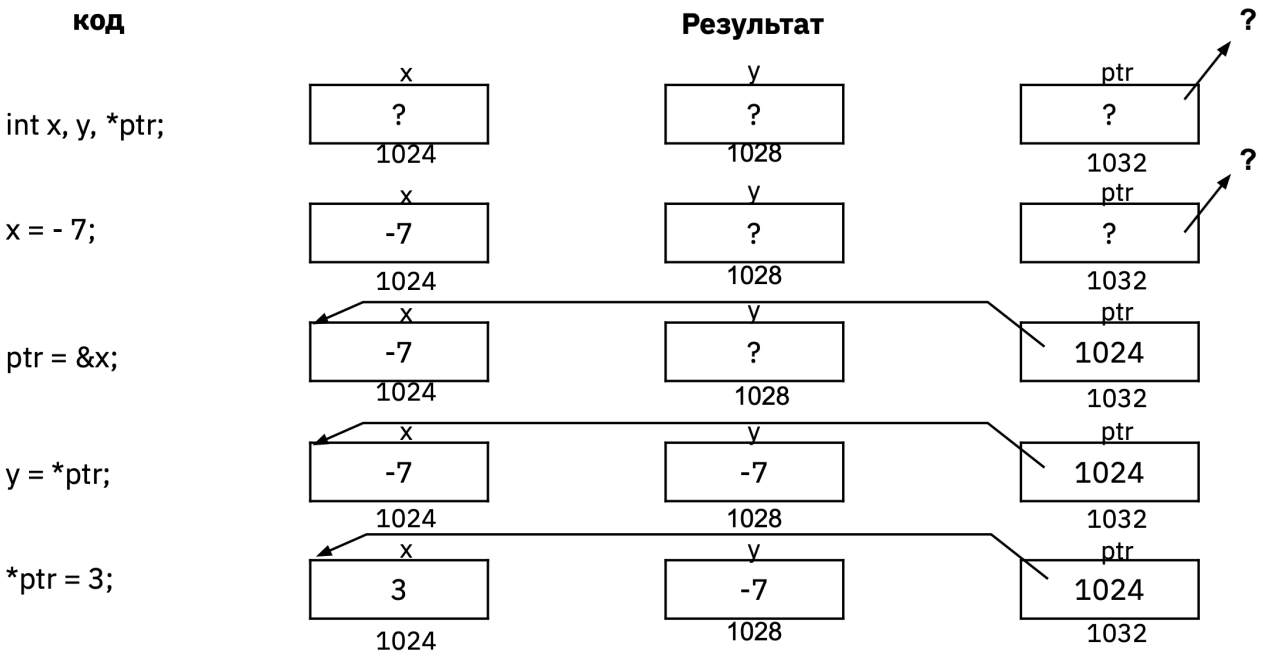
указывает указатель. В показанном ниже коде указатель использован для того, чтобы косвенным образом изменить содержимое переменной `x`.

```
int x, *px;
px = &x; // Инициализировать px адресом x
*px = 10; // Изменить значение, находящееся по адресу (т.е. значение x)
```

В данном примере оператор присваивания `*px = 10;` означает "присвоить `10` по адресу, содержащемуся в переменной `px`". Мы воспользовались косвенной адресацией. Итак, описаны две унарные операции над указателями: `&` и `*`. Приоритет этих операций тот же, что и других унарных операций языка Си.

Ещё одно, графическое, пояснение к операциям & и *:

Объявляем две целочисленные переменные x, y и указатель на целое ptr.



Компилятор расположил их по адресам `1024`, `1028`, `1032` (адреса только для примера). В переменных мусор, указатель никуда не указывает. Переменной `x` присваивается значение `-7`. Переменной `ptr` присваивается значение адреса переменной `x`, то есть, `1024`. Переменной y присваивается содержимое памяти, адрес которой (`1024`) находится в `ptr`, а содержимое ячейки памяти `1024` равно `-7`. Таким образом, теперь `y = -7;` Значение `3` заносится в ячейку памяти, адрес которой (`1024`) находится в `ptr`, а это адрес переменной `x`. Следовательно, `3` будет записано в `x`.

Указатели

Изучив операции `&` и `*`, мы переступили порог увлекательного и прекрасного мира работы с указателями, одного из самых мощных и выразительных инструментов языка C++. Указатели позволяют программисту работать с большими объемами данных, практически не перемещая их в памяти.

Как мы уже знаем, переменная – это символическое имя, определяющее некоторую область памяти для хранения информации в том или ином формате. Указатель – это тоже переменная, но ее значением является адрес области памяти, содержащей информацию. Как и любая другая переменная, перед использованием в программе указатель должен быть, как минимум, объявлен для компилятора, при этом указатель имеет свой собственный тип “указатель”, но созвучный с типом переменной, на которую он указывает. Вот так: `int*` это тип указателя на переменную типа `int`. Примеры такого объявления уже были приведены, но повторим:

```
int *xptr; // указатель на ячейку памяти, в которой содержится "целое"
char *cptr; // указатель на ячейку памяти, в которой содержится " символ"
```

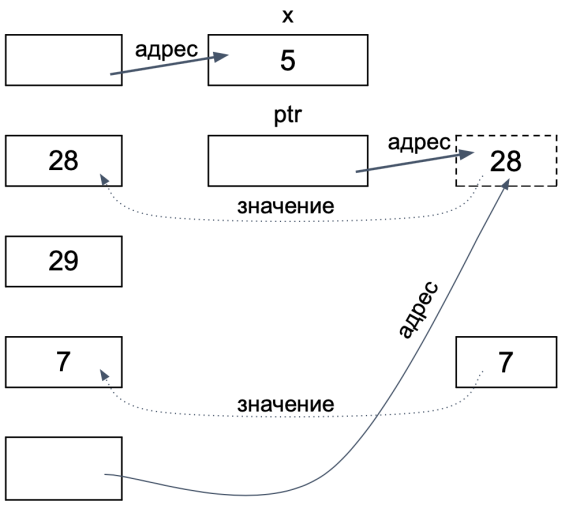
Мы уже знаем, что звездочка `*` является операцией косвенной адресации, операндом такой операции может быть любое выражение, результатом вычисления которого является указатель.
Хорошим тоном (но не обязательным правилом) является объявление указателя в той же самой строке, где объявляется переменная, на которую он указывает:

```
int x, *xptr; // обратите внимание на запятую в качестве разделителя.
              //Запись int x; *xptr; будет ошибочна!
char c, *cptr;
```

Имя указателя желательно выбирать созвучным имени переменной, на которую он указывает, как в нашем примере, чтобы самому потом не запутаться в именах. У нас `ptr` – это сокращение от слова pointer, что по английски значит указатель, а перед ним имя самой переменной, на которую этот поинтер указывает, но без всяких пробелов. Вот так:

```
int *x ptr; // будет ошибкой
int *x_ptr; // вполне допустимо.
```

Повторим, что хотя при объявлением указателя и стоит тип `int` или `char`, сам он имеет собственный тип (`int*` или `char*`), в зависимости от того, что находится в ячейке, на которую он указывает.
Как и для любой другой переменной, только лишь объявления указателя еще недостаточно, чтобы в нем содержался адрес ячейки, на которую он должен указывать. Для вычисления адреса памяти в C/C++ существует уже известная нам операция `&`.
Теперь покажем несколько операций для работы с указателями:

Операция	Результат	Представление в памяти
`&x`	Адрес переменной `x`	
`*ptr`	Значение по адресу, указанному в `ptr`	
`++(*ptr)`	Увеличить на единицу значение по адресу, указанному в `ptr`	
`*&y`	Значение по адресу, на который указывает `&y`, то есть само `y`	
`&*ptr`	Адрес области памяти, на которую указывает `ptr`, то есть само содержимое `ptr`	


```
int x = 5, y = 7, *ptr;  // переменные x, y проинициализированы по месту
объявления, указатель ptr только объявлен, но еще не проинициализирован,
указывает куда-то случайным образом
```

Пояснения к результатам двух последних операций:

- все вышеприведенные операции унарны. Унарные операции в С ассоциативны. Следовательно, порядок их исполнения таков (обозначен скобками):
- `*(&y);` т.е. сначала вычисляется `adr = &y;` т.е. адрес переменной `y;` затем вычисляется `*adr;` т.е. значение по адресу, указанному в `adr;` а по этому адресу находится переменная `y`
- `&(*ptr);` т.е. сначала вычисляется `*ptr;` результат – содержимое области памяти, на которую указывает `ptr;` затем операция `&` вычисляет адрес этой области, то есть, содержимое `ptr`

Замечания по поводу инициализации указателей:

```
// пусть:
int x, *xptr;
char c, *cptr;
// тогда:
cptr = &c; // Корректно. Теперь cptr указывает на c
*xptr = 5; // Ошибка! Сама по себе строка правильная, но так как
           // нигде до этого, начиная от строки int x,*xptr; указателю
           // xptr не было присвоено никакого значения, то это попытка
           // записать значение непонятно по какому адресу.
xptr = &x; // А вот так корректно,
*xptr = 5; // в этом случае переменной x присваивается значение 5.
xptr = cptr; // Ошибка! Оба указатели, но указывают на разные типы переменных
xptr = (int *) cptr; // Это пример явного преобразования типа указателя
xptr = 5; // Ошибка! Указателю нельзя присваивать никакое целое кроме NULL
xptr = 0; // Корректно, см.строку выше
// Еще несколько неочевидных ошибок при работе с указателями. Пусть:
float x, y, *ptr;
// тогда:
*ptr = &3.14; // Ошибка! Константа 3.14 не имеет адреса
*x = 7.5; // Ошибка! Переменная x объявлена не как указатель
x = &y; // Пример дурного тона и потенциальных ошибок в коде!
```

Переменная y объявлена как `float`, следовательно, `&y` имеет тип указателя, и компилятор обязан выполнить как операцию `&`, так и операцию присваивания `=`, но переменные слева и справа от `=` имеют разный тип. Компилятор, конечно, выдаст предупреждение о несоответствии типов при присваивании и пойдет дальше. Но кто же из "серьезных" программистов обращает внимания на такие мелочи. Строка будет откомпилирована, но поскольку `x` не указатель, потом, не в этой строке, а потом, в программе, с помощью `x` все равно невозможно будет добраться до данных, содержащихся в переменной `y`. Поэтому, все-таки, ошибка!

Арифметика указателей

Изменение значения указателя

Язык С++ допускает применение к указателям некоторых арифметических операций, но их выполнение отличается от выполнения тех же операций над числами. Эти отличия связаны с тем, что указатель на некоторый тип есть адрес размещения в памяти значений данного типа, т.е. размер занимаемой области есть `sizeof(тип)`. Таким образом, прибавление к значению указателя, например, единицы, приводит к реальному увеличению адреса на размер (обычно в байтах) элемента данного типа, т.е. к вычислению адреса памяти для следующего элемента данного типа.

```
тип *pt;
int x;
pt += i; // результат = увеличению pt на i*sizeof(тип). (* здесь умножение).
pt -= i; // Тип результата остается указателем на тот же тип.
pt++;
--pt;
```

Сравнение указателей

Операции сравнения ("операции отношения") также можно использовать без ограничений, когда указатели указывают на элементы одного и того же массива. В этом случае можно использовать даже вычитание указателей. А во всех остальных случаях сравнение ограничено только проверкой на равенство.

```
тип x, *px, *py;
// без ограничений:
if (px == py) ....;
if (px != py) ....;
// в случае одного и того же массива:
if (px > py) ....; // и другие операции сравнения
```

Указатели на массивы.

Имя массива есть константный указатель, поэтому его можно использовать в качестве указателя в выражениях, где указатель выступает в качестве rvalue. В том числе и в операциях косвенной адресации (`*`), и в операциях с индексированием `[]`.

```
// пусть:
int x[10], y[10], *ptr;
// тогда:
ptr = x; // это пример корректной инициализации указателя на массив.
         // Ранее мы говорили, что имя массива - это указатель на нулевой
         // элемент этого массива. Этот адрес и попадет в ptr
ptr = &x[0]; // корректно. Это эквивалент вышеприведенной записи.
x = x + 1; // Ошибка! Имя массива - это особый вид указателя - константный
           // указатель. Т.е. это rvalue и может встречаться только справа
           // от оператора присваивания.
int *xptr = x; // Корректно. Это просто еще один указатель на массив x[].
               // Объявление и сразу инициализация.
x = y; // Ошибка !!!
```

Хотя `x`, `y` имена массивов одного типа, имя `x` указывает на `x[0]`, а имя `y` на `y[0]`, оба имени `x`, `y` являются константными указателями, значит, ни одно из них не может встречаться слева от оператора присваивания. А вот конструкция `ptr = y;` вполне корректна, переменная типа указатель на `int` может быть переопределена в любой момент времени.

Указатели на символьные строки

Символьная строка есть массив символов. Поэтому операция присваивания не может быть использована для инициализации символьной строки вот таким образом:

```
char symbstr[10];
symbstr = "hello"; // Ошибка! Т.к. имя строки - это указатель-константа,
                   // и значение этого указателя не может быть изменено
```

Строковая константа (литерал `"hello"` в нашем примере) создается и поддерживается в C++ иначе, чем символьные и числовые константы, которые имеют постоянный размер и компилятор, соответственно, знает, сколько места отвести для их хранения в памяти. Строковая – же константа не имеет заранее известной фиксированной длины, поэтому компилятор должен обеспечить хранение всей строковой константы в памяти на все время работы программы. Поэтому в языке C++ значением строковой константы является адрес нулевого символа строки, а не текст самой строки, т.е. константный указатель, rvalue, как и для примера в указателях на массивы. Но можно объявить переменную типа указатель, и значение строковой константы присвоить этому указателю. То есть:

```
char symbstr[10];
char *ptr = symbstr;
```

```
ptr = "hello"; // корректно, и означает присваивание переменной ptr
               // значения адреса нулевого символа константной строки "hello"
```

Указатели на многомерные массивы.

Ранее мы говорили о том, что многомерный массив в языке Си трактуется как массив массивов. Вот пример объявления и инициализации двумерного массива:

```
#include<cstdio>
int main() {
int m[4][5] = {
 {8, -1, 17, 4, 20},
 {},
 {6, 11, 18, -4, 15},
 {}
}, *pm, *ppm; // pm и ppm оба объявлены как указатели на int
// примеры обращения к элементам такого массива с помощью указателей:
pm = m;      // Ошибка! Значение типа "int(*)[5]" нельзя присвоить "int *"
pm = m[0];   // Корректная инициализация указателя на массив
ppm = &m[0][0]; // Еще один указатель указывает на тот же массив, что и pm
pm = &m[0][4]; // указатель pm указывает на последний элемент нулевого
подмассива, т.е. на 20
printf("Hello. m[0][0] = %d; pm = %d; ppm = %d", m[0][0], *(pm-2), *(ppm+12));
return 5;
}
```

Получим в результате вывод

```
Hello. m[0][0] = 8; pm = 17; ppm = 18
```

А вот какие результаты даст применение операции sizeof к выражениям из этого примера (если размер int равен 4 байтам):

```
 sizeof(m) = 80;
 sizeof(m[0]) = 20;
 sizeof(m[0][4]) = 4;
```

Массив указателей.

Указатели могут быть объединены в массив как и любые другие переменные. Вот пример такого массива:

```
char *str_ptr[3] = {
 "Programming ",
 "is ",
 "fun!"};
```

Так как квадратные скобки `[]` более приоритетны, чем `*`, запись `*str_ptr[3]` аналогична записи `*(str_ptr[3])`.

Каждый элемент массива `str_ptr` в примере инициализирован адресом символьной строковой константы, а имя массива - это указатель на его нулевой элемент, поэтому нулевым элементом `str_ptr` будет указатель. Следовательно сам `str_ptr` - это указатель на указатель. Используя оператор косвенной адресации, запишем эти факты:

`str_ptr` - указатель на указатель на char;

`*str_ptr` - указатель на char; эквивалентный записи `str_ptr[0];` также эквивалентный записи `*(str_ptr + 0);`

`**str_ptr` - это символ. Это справедливо, т.к. `str_ptr[0][0]` - это символ, а запись `str_ptr[0][0]` эквивалентна записям: `*(str_ptr[0] + 0); *(str_ptr[0]); *str_ptr[0]; *(*str_ptr);` и, наконец: `**str_ptr;` Напомним, что `str_ptr` - константа, и операции (`++`) и (`--`) не применимы к ней.
Используется массив указателей аналогично массиву массивов, к отдельному элементу можно обратиться либо с помощью индексов, либо через указатели. Однако между ними есть и существенные различия:

- они по разному отображаются в памяти;
- `str_ptr` есть указатель на указатель на `char`, т.е. его тип `char**`.

Указатели на указатели

Собственно, мы уже ввели понятие указателя на указатель при обсуждении массива указателей. Теперь введем понятие типа переменной указатель на указатель. Вот как объявляется и инициализируется такая переменная (продолжим пример из раздела массив указателей):

```
char **ptr_ptr;
char **ptr_ptr = str_ptr;
```

после такого объявления и инициализации переменная `ptr_ptr` имеет то же значение, что и `str_ptr`, и указывает на ту же область памяти - нулевой элемент массива `str_ptr`. Но в отличие от `str_ptr`, `ptr_ptr` - это элемент типа указатель на указатель, а не константный указатель типа имя массива. Поэтому, незначительно забегая вперед, в занятия по операторам языка и библиотечным функциям, покажем такой пример:

```
for (i = 0; i < 3; i++) printf("%s", ptr_ptr++);
```

После объявления и инициализации `ptr_ptr` в этом разделе корректно распечатается строка "Programming is fun!", а вот такое выражение, при полной внешней схожести:

```
for (i = 0; i < 3; i++) printf("%s", str_ptr++);
```

Вызовет ошибку, поскольку `str_ptr` - это константный указатель, и к нему неприменима операция `++`

Операция указателя на структуры

Операция `->` требуется в том случае, если для ссылки на элемент структуры используется указатель.

```
struct data {
  int day;
  int month;
  int year;
}; // Объявление структуры
struct data New_Years_Day, *data_ptr; // Объявление переменной типа объявленной
структуры и указателя на эту переменную
*data_ptr = New_Years_Day; // Инициализация указателя на переменную,
проинициализировать саму переменную New_Years_Day можно так:
(*data_ptr).day = 1; // скобки необходимы, т.к. приоритет * ниже приоритета .
(*data_ptr).month = 1;
(*data_ptr).year = 21;
// а можно так:
data_ptr->day = 1;
data_ptr->month = 1;
data_ptr->year = 21;
```

Записи полностью идентичны, но второй вариант записи содержит намного меньше символов операций и читается намного легче.

Практическое задание

1. Написать программу, вычисляющую выражение a * (b + (c / d)) и выводящую результат с плавающей точкой, где a, b, c, d – целочисленные константы;
2. Дано целое число, выведите на экран разницу между этим числом и числом 21. Если же заданное число больше, чем 21, необходимо вывести удвоенную разницу между этим числом и 21. При выполнении задания следует использовать тернарный оператор.
3. * Написать программу, вычисляющую выражение из первого задания, а переменные для неё инициализировать в другом файле
4. * Описать трёхмерный целочисленный массив, размером 3x3x3, указатель на значения внутри массива и при помощи операции разыменования вывести на экран значение центральной ячейки получившегося куба [1][1][1]

Используемые источники

1. *Брайан Керниган, Деннис Ритчи.* Язык программирования C. — Москва: Вильямс, 2015. — 304 с. — ISBN 978-5-8459-1975-5.
2. *Stroustrup, Bjarne* The C++ Programming Language (Fourth Edition)