Лекция 6. Переменные Ввод - вывод в С

Переменные

Чтобы завести переменную, мы должны определить ее имя, тип и значение, то есть переменная = тип + имя + значение. Каждая такая переменная является частью программы. Стоит отметить, что *ключевые слова* не могут быть именами переменных. Объявление переменной происходит следующим образом: type name [, name, name]; Можно задать класс памяти и начальное значение переменной int a, b; unsigned c = 2019. Ключевые слова – это служебные слова: базовые типы, встроенные в язык, ряд квалификаторов, слова, указывающие на операторы.

Перед тем как написать инициализацию переменной, определим, где мы можем пользоваться этой переменной, то есть определим область действий (scope).

Переменная может быть объявлена

1. Внутри функции или блока (локальная)
2. В объявлении функции (параметр функции)
3. Вне всех функций (глобальная)

Часто класс памяти переменной (storage) путают с областью действий переменных, потому что некоторые области действий автоматически влекут за собой определенный класс памяти, но это не всегда правда. Область действий

- *Локальной переменной* – блок, в котором она объявлена (C99 – начиная со строки объявления)
- *Глобальной переменной* – программный файл, начиная со строки объявления

В одной области действий нельзя объявлять более одной переменной с одним и тем же именем. Чтобы в языке программирования не возникал конфликт имен, можно всегда либо запретить это, либо сделать действие по умолчанию. Например, есть функция с двумя переменными x: глобальной и локальной

```
int x
    {
    int x
        x + y

    }
```

Чтобы здесь разрешить противоречие, нужно взять ближайшую область действий, то есть используется именно локальная переменная.

В таких случаях компилятор может выдавать предупреждение (warning Wshadow), на которые стоит обращать внимание.
Рассмотрим пример, который все это демонстрирует. У нас есть некоторая глобальная переменная count, ее областью видимости является весь файл. В функции func также появляется локальная переменная count. Присваивание, которое отнимает 2 из переменной count глупое, поскольку это действие никогда за пределы функции не выйдет.

```
#include <stdio.h>
int count;          /* global */
void func (void) {
        int count;      /* auto */
        count = count - 2;
}
static int mult = 0; /* static */
int sum ( int x, int y) {
        count ++;
        return (x + y) * (++mult);
}
int main(void) {
        register int s = 0; /* register */
        count = 0;
        s += sum(5, 7);
        func ();
        printf ("Sum is %d func is called %d times\n", s, count);
        return 0;
}
```

Отметим, что к глобальной переменной могут обратиться разные функции, раз разные функции могут обратиться к этой переменной, то компилятор не может управлять ее временем жизни, то есть она не может быть автоматической. Эта переменная должна жить какое-то время, в течение которого работают функции. Но какие функции работают точно неизвестно, поскольку в С раздельная компиляция. То есть можно объявить переменную таким образом, что с ней смогут работать функции из другого файла. Поэтому эту переменную мы вынуждены заводить на время работы всей программы, то есть она попадает в статическую память.

С локальными переменными ситуация обратная. По умолчанию локальная переменная – способ завести ячейку в автоматической памяти. Так в функции func компилятор выделит память для переменной count. Он знает ее размер и тип. Но можно нарушить это умолчание и завести переменную в статической памяти. Для этого есть квалификатор static.

Например,

```
int foo() {
        static int x = 1;
        .
        .
        .
        x= x + 1
}
```

Переменная x локальная, за пределами функции foo ей нельзя воспользоваться. Квалификатор static показывает, что переменная x хранится в статической памяти. По сути, x + x+1 означает, что переменная x будет счётчиком вызовов этой функции. Поскольку это статическая переменная, то после работы функции она никуда не исчезнет и значение тоже останется. При первом вызове функции x = 1, при втором x =2 и так далее. Инициализация же выполнится только один раз, перед началом программы.

Инициализация переменных

В зависимости от класса памяти и нахождения переменной инициализация выполняется по-разному. Так, при объявлении переменной int x = 42;

- Автоматические переменные инициализируются каждый раз при входе в соответствующий блок;
- Если нет инициализации, значение соответствующей переменной нее определено!
- Глобальные и статические переменные инициализируются только 1 раз в начале работы программы;
- Если нет инициализации, они обнуляются компилятором*;
- Внешние переменные инициализируются только в том файле, в котором они определяются;
- При инициализации переменной типа с квалификатором const она является константой и не может менять свое значение.

*Если нет инициализации переменные зануляются. Если регистровый класс памяти, выдаем инструкцию для записи 0 в регистр. Если у нас статический класс памяти, то в ко, который выполняется до начала работы запишем, чтобы статическая память была занулена. А у автоматической памяти свои нюансы. При вызове функции после выделения автоматической памяти нужно записать 0 на всю автоматическую память. Кроме того, это действие должно выполнятся каждый раз при вызове функции. А

функция может вызываться много раз. Решение следующее: если инициализации нет, то автоматическая память не инициализируется, то есть в ней хранится мусор. Поэтому автоматическую память всегда нужно инициализировать самому. Если же все-таки мы прочитаем, что находится в неинициализированной переменной, то столкнемся с неопределенным поведением (undefined behavior).

Компилятор может предупредить, когда глобальная и локальная переменные имеют одинаковое имя. Однако понять всегда ли переменная инициализируется в функции – алгоритмически неразрешимая задача.

Как записывать константы? Инициализация – по сути, присваивание переменной некоторого заранее заданного неизменяющегося значения, то есть константы. Литералы задают константу (фиксированное значение). При этом:

- Символьные константы задаются символом или его кодом в кавычках: ‘c’, ‘L’, ‘\0x4f’, ‘\040’. Тип символьной константы - int!

- Целые константы 100, -34l, 1000U, 999llu, u- unsigned, l (L) – long, ll- long long, порядок суффиксов роли не играет.
- Константы с плавающей точкой 11.123F, 4.56e-4f, 1.0, -11.123, 3.1415926l, -6.62068e-34L. Тип вещественной константы без суффикса f (F) – double! Можно опускать либо целую, либо дробную часть опускать, также запись с мантиссой.
- Шестнадцатеричные константы 0x80 (128). Префикс такой константы 0x.

  Вещественные шестнадцатеричные: 0x3.ABp3 ($3\frac{171}{256} \times 8 = 29.34375$
- Восьмеричные константы 012 (10). Префикс – 0.
- Строковые константы “a”, “Hello, World!”, L“Unicode string”
- Специальные символьные константы (управляющие последовательности) \n, \t, \b

Операции над целочисленными данными

Операции над целочисленными данными обычно группируются по местности, то есть по количеству операндов. Есть операции с 1, 2 и 3 операндами.

Арифметические операции

Одноместные: изменение знака (-), одноместный плюс (+)
Двухместные: сложение (+), вычитание (-), умножение (*), деление нацело (/), остаток от деления нацело (%)

(a / b) × b + (a % b) = = a – формула для определения остатка, остаток может быть меньше 0.

Отношения (результат 0/1 типа int)

Больше (>), больше или равно (> =), меньше (<), меньше или равно (<=)

Сравнения (результат 0/1 типа int)

Равно (= =), не равно (! =)

Логические операции

Отрицание (!), конъюнкция (&&), дизъюнкция (| |), ложное значение – 0, истинное значение – любое ненулевое
«Ленивое» вычисление && и | | означает, что если нужно вычислить выражение с конъюнкцией или дизъюнкцией, то операнды этого выражения вычисляются слева направо, вычислив первое выражение, уже можно предсказать результат операции. Например, если первое выражения равно 0, то второе вычисляться не будет. Таким образом у нас есть гарантия, что первое выражение вычислено. Это можно использовать так:

```
if a != 0  && b/a > 3 {
}
```

Если a = 0, то результат конъюнкции уже известен и b/a вычислено не будет, а если бы оно вычислялось, то мы бы делили на 0, что выдало бы ошибку.

Операции присваивания

В С есть понятие побочного эффекта (side effect). Побочный эффект заключается в изменении памяти, изменении объекта. Присваивание вводит в язык термины «левый» и правый» операнд присваивания, lvalue и rvalue соответственно, описывающие выражения, которые могут стоять слева и справа. lvalue = rvalue.

- lvalue – выражение, указывающее на объект памяти
- rvalue – выражение, генерирующее значение

Присваивание – это не оператор, а выражение, которое генерирует значение, правую часть, rvalue. Это не то же самое, что просто написать rvalue, у присваивания есть побочный эффект, который заключается в том, что в память, на которую указывает lvalue, заносится значение rvalue. Например, в выражении a = 0 значением является 0, а

побочным эффектом является то, что в a будет записан 0. Из-за того, что у выражения есть значение, мы можем писать цепочки. Пример: a = b = c = d = 0;
У них ассоциативность справа налево.

Укороченное присваивание: lvalue op = rvalue, где op -двухместная операция.
Пример: a += 15;
Укороченное присваивание построено на следующей идее: часто адрес результата совпадает с одним из операндов и можно результат записывать в этот операнд и таким образом сэкономить 1 адрес. Если нужно не испортить один из операндов, тогда можно сохранить значение в один из регистров. Таким образом, в выражении a = a + 2 a вычисляется 2 раза, а в выражении a += 2 – только один раз.

Инкремент и декремент тоже реализованы и действуют по логике двухместных операций. Когда люди писали ассемблер, они заметили, что частая операция сложения или вычитания — это сложение (вычитание) с 1. Поэтому для этого случая была придумана своя мнемоника. В ассемблере очень часто бывает инструкция inc и dec. В C это реализовано как ++ и -- соответственно. Инкремент (декремент) означает, что значение операнда должно быть увеличено (уменьшено) на 1 и записано обратно в операнд.
Если операция требует записи операнда, то этот операнд должен быть lvalue.
Назревает вопрос, что наступает раньше побочный эффект или вычисление выражения в записи a++? В выражении a = a + 1 такой проблемы нет, так как сначала происходит вычисление выражения (rvalue), а затем побочный эффект. Заметим, что побочный эффект в a++ и a = a+1 одинаковый. Можно записать выражение с ++ как в префиксной, так и в постфиксной форме. Выражение a++ генерирует значение до инкремента (верни a и увеличь на 1), а выражение ++a генерирует значение после инкремента (увеличь на 1 и верни a). Это легко запомнить, если считать, что операции выполняются слева направо. Например, если a = 3, то a++ вернет 3, в то время как ++a вернет 4.

Также есть двухместный операнд “,” для последовательного вычисления.
Пример: a = (b = 5, b+2);
Операция « , » вычислила операнд b = 5,у которого есть значение 5 и побочный эффект, который заключается в том, что в b записали 5. Далее вычисляем правый операнд, а побочный эффект уже произошел. Результат операции « , » - результат правого операнда, то есть в результате значение станет равно 7.

В языке Си нет никаких ограничений, в каком порядке будут вычисляться операнды. Например, в выражении a*2 + b/3 никто не гарантирует, что сначала будет выполняться умножение, может быть наоборот. В функции с большим количеством аргументов, аргументы могут быть вычислены в удобном нам порядке. То есть у компилятора есть полная свобода менять порядок вычислений операндов так, как ему покажется быстрее.

Однако есть некоторые ограничения. Эти ограничения записываются через *точку следования* (sequence point).
Точки следования – это, по сути, моменты, когда мы знаем, что все случилось и transaction completed. Пример из жизни: пришла смс, что деньги списались, пока не пришла смс непонятно, что случилось. Аналогично точки следования описывают момент, в который должны приходить смс, глядя на которые мы понимаем, сколько осталось денег.

Формально: *точка следования* – момент, во время выполнения программы, в котором все побочные эффекты предыдущих вычислений закончены, а новых – не начаты. В этот момент в rvalue мы можем быть уверены, что lvalue уже записал все в память. Ленивые вычисления обязаны влечь за собой точку следования.
Самый популярный способ получить в коротком выражении точку следования конъюнкцию или дизъюнкцию (&& или | |). Из-за ленивого вычисления, из-за того, что всегда сначала вычисляется левая часть, а потом правая, здесь наступит точка следования. При этом между двумя точками следования изменение значения переменной возможно не более одного раза.
Второй способ – это то, что называется full expression, самое длинное выражение, которое написали до ;. То есть c = d, это не full expression, a a = b = c = d = 0; - full expression. Это место, когда expression - операция в терминах стандарта превращается в expression statement, то есть в оператор, для этого нужно поставить ;. Поставили точку-произошел sequence point.
Напомним, что при вызове сложных функций с множеством аргументов, как, например, f(g(5), h(3)) вовсе необязательно, чтобы первой вычислялась g(5), может быть и наоборот.
В примере i++ + ++I нет точек следования, поэтому нельзя менять один и тот же объект 2 раза, это приведет к undefined behavior. Результат этого выражения будет зависеть от компилятора и его поведения.
В последних стандартах терминология несколько иная (sequenced before, unsequenced, indeterminately sequenced): точка следования влечет частичный порядок, его отсутствие делает возможным любые варианты.

Форматный ввод-вывод

Пока у нас нет указателей, мы фактически не можем ничего ввести. Есть функция scanf, в которую нужно передать указатели на числа], то есть их адреса, если мы хотим их считать два числа. Адреса передаются с помощью оператора &.

```
#include <stdio.h>
int main(void) {
        int s = 0;
```

```
        int a, b;
        sacnf("%d%d", &a, &b);
        s += a + b;
        printf ("Sum is %d\n", s);
        return 0;
}
```

У нас есть соглашения: стандартный поток ввода (stdin) и стандартный поток вывода (stdout). Эти функции читают стандартный поток ввода, который для нашего удобства является клавиатурой, и стандартный поток вывода, который является монитором. Также есть еще стандартный поток ошибок.
Чтобы ввести число типа int, например, мы могли бы написать функцию, но тогда для числа типа double пришлось бы писать свою функцию, что очень неудобно, особенно если учесть, что в С нет перегрузки имен, как в С++, и мы не можем одним и тем же именем назвать несколько функций.

В Си ввод-вывод реализован с помощью форматный строк, то есть мы подсказываем компилятору, какие типы данных мы считываем и записываем, зная это, компилятор может реализовать свой ввод-вывод для каждого типа. Форматный ввод-вывод так называются, потому что то, что мы считываем или записываем, определяется форматом, а формат определяется форматной строкой.

В нашем примере функция scanf – это первая функция с переменным количеством параметров, с который мы встретились. Мы можем считать как одно число, так и несколько чисел. Первым параметром обязана быть форматная строка, которая описывает, что мы считываем. % — это форматный символ (спецификатор), то, что идет после него — это тип того, что мы считываем или записываем.

Спецификатор	Печатает/считывает
%d, %ld, &lld	Число int, long, long long
%u, %lu, %llu	Число unsigned, unsigned long, unsigned long long
%f, %Lf	Печатает double, long double
%f, $lf, %Lf	Считывает float, double, long double
%c	Символ (char)

%4d: вывести число типа int минимум в 4 символа

%.5f: вывести число типа double с 5 знаками
%%: напечатать знак процента
Функция scanf возвращает количество удачно считанных элементов.