Слайд 2

Приведение типов

В языке Си не только переменные, но и константы имеют тип. Например, константы: `3` и `-7` – это целочисленные константы, а `3,14` и `-0,012` – константы с плавающей запятой. По умолчанию, тип целочисленной константы – `int`, а тип константы с плавающей запятой – `double`. Любое выражение тоже имеет тип – это тип результата, получающегося после вычисления этого выражения.

Что же произойдет, когда мы значение переменной одного типа присвоим переменной другого типа? Или если в одном выражении смешать переменные и константы разных типов? В языке Си, как и в других языках программирования, все двухместные операции умеют работать только в том случае, когда оба их операнда имеют одинаковый тип. Поэтому, отвечая на вопрос – произойдет то, что в языке Си называется *приведением типов данных* (или преобразованием типов).

Преобразования бывают *явными* и *неявными*. Неявные преобразования производит сам компилятор без помощи программиста. Они происходят везде и всюду в наших программах и их можно разделить на два класса:

1. когда мы присваиваем результат вычисления какого-либо выражения переменной – этот результат преобразуется к типу переменной;
2. при вычислении выражения операнды каждой операции преобразуются к какому-то одному типу (если они различаются), при этом результат операции будет того же типа.

Во втором случае основное правило неявного преобразования двух типов к какому-то одному: *компилятор всегда старается преобразовать «меньший» тип к «большему», чтобы минимизировать возможную потерю информации*. Выпишем все базовые типы в порядке возрастания их «размера»: `char` ≤ `short` ≤ `float` ≤ `int` ≤ `long` ≤ `double`. Если в результате операции возможна потеря точности, компилятор выдаст соответствующее предупреждение, однако останавливать процесс компиляции из-за этого не будет, так как, строго говоря, это не считается ошибкой.

Слайд 3

Рассмотрим подробней два класса неявных преобразований. К первому классу, как уже было сказано, относятся преобразования, происходящие при присваивании. В этом случае правило преобразования «меньшего» типа к «большему» не действует. Результат вычисления выражения, стоящего справа, преобразуется к типу той переменной, которая стоит слева. Например:

```
int a = 2.3 + 4.5;
```

Очевидно, что результат вычисления выражения будет дробным числом, но вот переменная, в которой результат сохраняется – целочисленная. Поэтому происходит преобразование: дробный

результат `6.8` преобразуется к типу `int`, причем преобразование будет заключаться в том, что дробная часть отбросится. Таким образом, в предыдущем примере в переменной `a` сохранится значение `6`. Еще раз обратим внимание, что при преобразовании типа `float` или `double` к любому целочисленному типу *никакого округления не происходит* – дробная часть просто отбрасывается, что и называется компилятором *потерей данных,* о которой он сообщит вам в соответствующих предупреждениях. Во время преобразования целочисленного типа к дробному тоже может произойти потеря данных. Здесь играет роль количество значащих цифр дробного типа, которое обсуждалось выше. Например, тип `double` сохранит без потерь любое число типа `int`, а вот тип `float` – только часть диапазона.
Второй класс преобразований – преобразования, происходящие внутри выражения в момент его вычисления. Рассмотрим пример:

```
float a = 5 / 6;
```

Это выражение, состоящее из одной операции деления. Операция видит, что оба его операнда целочисленные, поэтому ничего преобразовывать не нужно и, как следствие, *результат тоже будет целочисленным*. В этом примере результатом целочисленного деления `5` на `6` будет ноль. Далее этот целочисленный ноль преобразуется к типу с плавающей запятой и записывается в переменную, но, увы, уже поздно – информация о дробной части потеряна. Для того чтобы исправить эту ситуацию, достаточно, например, изменить тип хотя бы одного из операндов на дробный путем добавления десятичной точки:

```
float a = 5.0 / 6;
```

В этом случае, как уже отмечалось ранее, должно будет выполниться преобразование типов, чтобы получились однотипные операнды. Тип константы `5.0` – `double`, тип константы `6` – `int`, и компилятор выбирает самый благоприятный для сохранения информации вариант – преобразует константу `6` к типу `double`. Соответственно, и результат вычисления выражения будет тоже дробным.

Слайд 4

Схема, описанная выше, действует в любом выражении. Например:

```
int a, b;
double f1, f2;
f1 = a / (f2 + b) - b;
```

Выражение вычисляется согласно приоритетам операций шаг за шагом: от самой глубокой (приоритетной) операции к самой внешней, при этом операции одного приоритета выполняются слева направо. В примере выше сначала выполнится сложение, затем деление, и только потом – вычитание. Каждая из этих операций будет приводить оба своих аргумента к одинаковому типу в соответствии с описанными выше принципами. В примере выше:

1. сложение преобразует тип значения переменной `b` к `double` и выдаст результат типа `double`,
2. деление преобразует значение `a` тоже к типу `double`, так как результат только что вычисленного второго операнда будет типа `double`,
3. наконец, вычитание сделает ровно то же самое, так как первый операнд будет иметь тип `double`.

В результате значение всего выражения также будет иметь тип `double`.

Слайд 5

Однако, здесь таится одна опасность. Например, чему будет равно значение выражения `2 / 4 * 3.14`? Казалось бы, `1.57`. Так как деление и умножение – это операции одного приоритета, то выполняются они слева направо. Поэтому сначала выполняется деление, а потом уже умножение. Но оба операнда деления – целочисленные! Значит и деление будет тоже целочисленным с результатом `0`. Оператор же умножения преобразует свой первый операнд к типу `double`, но будет уже поздно. В результате вычисления всего выражения мы получим ноль. Решить эту проблему можно, например, записав вместо целочисленной `2` дробную `2.0`.

На все эти вопросы стоит обращать особое внимание, так как по невнимательности в результате неявных преобразований можно случайно потерять значимую информацию, что иногда приводит к трудноуловимым ошибкам.

Слайд 6

Но что же нам делать, если подобная проблема возникает в том случае, когда в выражении содержатся переменные вместо констант? Например:

```
int a = 1, b = 2;
double result = a / b;
```

Написать `a.0`, как это мы делали в случае с константами, мы не можем! Здесь нам поможет *явное преобразование типов*, с помощью которого мы явно скажем компилятору, что мы хотим преобразовать значение одной из этих переменных к типу `double`:

```
int a = 1, b = 2;
double result = (double)a / b;
```

Значение переменной, идущей после скобок, будет преобразовано к типу, указанному в скобках. Явно можно преобразовывать тип не только значений переменных, но и целых выражений:

```
float a = 4.0, b = 2.0;
int result;
result = (int)((a * a) / (b * 6) + 4);
```

Слайд 7

Программа на языке Си состоит из операторов, заканчивающихся точкой с запятой. Самым простым из них является *пустой оператор*, состоящий только из точки с запятой:

```
;
```

который ничего не делает. Еще одним оператором, с которым мы уже познакомились, является *присваивание*. Идущие друг за другом операторы называются *последовательностью операторов*, что является первой базовой конструкцией любого структурированного языка программирования. При этом программа выполняется именно в этой последовательности – прямолинейно, оператор за оператором:

```
<оператор 1>;
<оператор 2>;
…
<оператор N>;
```

Слайд 8

Язык Си допускает запись операторов в одну строку, однако лучше этого избегать, так как это приводит к плохо читаемому коду.

Операторы можно объединять в группы (блоки) с помощью фигурных скобок, создавая тем самым *сложные (составные) операторы*:

```
{
    <оператор 1>;
    <оператор 2>;
    …
    <оператор N>;
}
```

Ключевой особенностью блоков является то, что они могут быть сколь угодно вложенными друг в друга. В языке Си блоки могут существовать сами по себе, в то время как в некоторых других языках они используются только вместе с ветвлениями и циклами. Сами блоки кода не добавляют ничего нового в логику исполнения программы – она по-прежнему выполняется последовательно: оператор за оператором, блок за блоком.

Слайд 9

Если бы мы были ограничены возможностью писать только последовательно выполняющиеся программы, то это бы существенным образом сузило класс решаемых задач. По сути, это сделало бы такой язык практически бесполезным. Поэтому в языке Си, как и в любом другом структурированном языке программирования, помимо конструкции последовательного выполнения операторов, есть еще две фундаментальные конструкции: *ветвление* и *цикл*.

Ветвление используется тогда, когда возникает необходимость изменить ход выполнения программы добавлением развилки, разбивающей его на две и более альтернативных ветви. Синтаксис конструкции ветвления следующий:

```
if (условие) <оператор>
```

Условием является *любое* выражение. Как правило, применяется логическое выражение, но язык Си позволяет использовать и любое арифметическое, трактуя его нулевое значение как ложь, а любое отличное от нуля – как истину. Логика выполнения конструкции очень простая – *если условие истинно, то выполняется идущий следом <оператор>, в противном случае он пропускается*.

В качестве <*оператор*> может быть любой оператор: *одиночный* (если значение `a` отрицательно, то мы его делаем положительным):

```
if (a < 0) a = a * -1;
```

составной (если `a` больше `b`, то меняем значения переменных местами):

```
if (a > b) {
int tmp = a;
a = b;
b = tmp;
}
```

и даже *пустой* (это допустимо, но вы должны хорошо понимать, зачем вам это нужно):

```
if (a % 2);
```

Всюду далее, если не оговорено иное, под `<оператор>` мы будем понимать любой из перечисленных выше типов операторов.

Обратите внимание на условие в последнем примере. Это арифметическое выражение, возвращающее `0`, если значение переменной `a` четное, и `1` – если нечетное. Таким образом, условие истинно, если переменная `a` нечетная.

Конструкция `if (условие)` «цепляется» к одному из операторов некоторой последовательности, делая его условным, то есть зависящим от выполнения определенного условия. Поэтому будет не совсем корректно называть саму конструкцию `if` оператором и ожидать, что после нее мы тоже должны поставить точку с запятой. Точка с запятой является частью того оператора, к которому «прицеплена» конструкция ветвления, именно поэтому она отсутствует после закрывающей фигурной скобки, потому что после фигурной скобки, ограничивающей блок кода, точка с запятой не ставится.

Иногда бывает необходимо указать альтернативный код, который будет выполнен тогда, когда условие ложно. Для этих целей служит else-секция ветвления:

```
if (условие)
<оператор_1>
```

```
else
<оператор_2>
```
Логика работы этой конструкции следующая: *если условие истинно, то выполняется <оператор_1>, а <оператор_2> пропускается, а если условие ложно, то все наоборот: <оператор_1> пропус-кается, а <оператор_2> выполняется*. Рассмотрим пример.

Пример 1 Написать программу, спрашивающую у пользователя два числа и выводящую «TRUE», если первое меньше второго. В противном случае вывести «FALSE».

Решение:

```
int a, b;
printf("Введите два числа: ");
scanf_s("%d %d", &a, &b);
if (a < b)
printf("TRUE\n");

else
printf("FALSE\n");
```

Пример 2 Написать программу, спрашивающую у пользователя три числа и выводящую наименьшее и наибольшее из них.

Решение:

```
int a, b, c, max, min;
printf("Введите три числа: ");
scanf_s("%d %d %d", &a, &b, &c);
if (a >= b && a >= c){
max = a;
}
else {
if (b >= a && b >= c)
max = b;
else
max = c;
}
if (a <= b && a <= c){
min = a;
}
else {
if (b <= a && b <= c)
min = b;
else
min = c;
}
printf("Max = %d, min = %d\n", max, min);
```

Иногда встает задача выбора одного из множества вариантов продолжения программы в зависимости от значения некоторого арифметического выражения. Это можно реализовать с помощью нескольких условных операторов, но удобнее воспользоваться оператором switch (рис. 2.3). Конструкция этого оператора включает заголовок в круглых скобках и тело в фигурных скобках, которое размечается с помощью ключевого слова case с различными целочисленными константами. В начале выполнения оператора switch вычисляется арифметическое выражение в его заголовке и далее происходит переход на метку case с тем же значением. Если значение заголовка не совпадает ни с одним из значений меток case, то либо происходит переход на метку default, либо, если такой метки нет, ни один из операторов тела switch не выполняется и переход происходит на следующий за switch оператор. Обратите внимание, что каждый case на схеме завершается оператором break, который завершает выполнение оператора switch. Его использование не является обязательным, но именно он позволяет реализовать блок-схему слева. В противном случае после перехода на одну из меток case далее продолжилось бы выполнение всех оставшихся до закрывающей фигурной скобки операторов. Иными словами, вход в тело оператора switch происходит по месту расположения соответствующей метки case, а выход либо с помощью оператора break, либо после выполнения последнего оператора тела. Ниже приведён пример использования оператора switch