Тарасов О.В., Южанин В.В.

Обзор языка С

Введение

В этой главе мы дадим краткий обзор языка С применительно к программированию микроконтроллеров. За многие годы было написано большое количество книг по этому языку, поэтому нет смысла писать ещё одну, подробно рассматривая каждую мелочь. Вместо этого, в начале каждого раздела будет приводиться небольшой пример, демонстрирующий тот или иной аспект применения С, после чего этот пример будет подробно разобран и объяснён. Таким образом, с первой же страницы мы начнём писать реальные программы для микроконтроллера, не забивая голову теоретическим материалом, который может никогда не пригодиться.

Первая программа

Традиционно при изучении нового языка программирования первой программой становится одна-единственная строка, выводящая на экран сакральное “Hello world!“. Однако у микроконтроллера нет ни экрана, ни вообще какого-либо стандартного средства вывода. Поэтому сейчас мы ограничимся написанием чуть более сложной программы, состоящей из целых 11 строк.

```
int main()
{
	int a = 2;
	int b = 3;
	int c = 1;
	ind D;

	D = (b * b) - (4 * a * c);

	return 0;
}
```

Эта небольшая программа находит дискриминант квадратного уравнения при заданных величинах *a***,** *b* и *c*. Конечно, она не несёт никакой практической ценности, так как значения «зашиты» в программу на этапе компиляции, однако, в качестве примера эта программа позволит нам понять некоторые ключевые концепции языка С.

Разберём этот код построчно. На первой строке мы видим объявление **функции**. Функция — это основной «строительный блок», из которых собирается программа. Она представляет собой отдельный участок кода, к которому можно обратиться по имени из другого места программы. Перед именем функции *main* расположен **тип возвращаемого ей значения**. На данный момент не будем заострять на этом внимание, и вернёмся к этому вопросу позже, при более подробном рассмотрении функций. Скажем лишь, что любая функция имеет вид:

```
Тип	имя_функции([список_параметров])
{
	Код_функции
}
```

Код, который непосредственно выполняется функцией, заключается в фигурные скобки *{ }*. Эти скобки используются для выделения любого логического блока в программе, и мы ещё вернёмся ним при рассмотрении других примеров.

В самом начале блока, обозначенного фигурными скобками, расположены **объявления переменных**. Переменная — это просто участок памяти, к которому мы можем обратиться по имени. Размер этого участка определяется **типом** переменной. Например, если мы объявим переменную *int foo*, в памяти будет выделен блок размером два байта — именно столько занимает тип данных *int*. Теперь, если мы будем обращаться в коде к переменной *foo*, компилятор поймёт, что ему необходимо обратиться к определённой области памяти, выделенной именно под эту переменную.

Таким образом, тип переменной отвечает за то, сколько памяти будет выделено на хранение информации, содержащейся в этой переменной. Стандарт С определяет несколько основных типов данных.

Тип данных	Размер выделяемой памяти	Описание
char	1 байт	Обычно используется для хранения символьных значений
int	2 байта	Целочисленный тип
float	4 байта	Число с плавающей точкой
double	8 байт	Число с плавающей точкой двойной точности

Существует также некая абстрактная величина, называемая *словом*. Размер слова исчисляется в битах или байтах и определяет разрядность основных регистров процессора, максимальный объём адресуемой памяти и ещё много интересных вещей. Тип данных *int* обычно равен размеру слова на конкретной платформе. В случае 8-битных контроллеров ATMega слово занимает два байта, столько же занимает и *int*. Все эти сложные слова были сказаны для того, чтобы подчеркнуть тот факт, что тип *int* является «родным» для контроллера, и именно операции над переменными этого типа производятся быстрее всего.

Стоит заметить, что при программировании микроконтроллеров типы данных с плавающей точкой используются довольно редко, так как операции с ними отнимают слишком много ресурсов. Для работы с дробными значениями чаще применяют числа с фиксированной точкой, реализуемые при помощи переменных типа *int* и побитового сдвига.

Четыре базовых типа можно расширить с помощью *спецификаторов*. Спецификатор — это ключевое слово, которое располагается перед базовым типом переменной. Существует 4 спецификатора: *signed, unsigned, short, long*. Первые два влияют на диапазон значений, принимаемых переменной, а два других — на размер памяти, выделяемой для переменной.

Количество байт, выделяемых для переменной, влияет на диапазон значений, которые эта переменная может принять. Например, для типа *char* выделяется один байт памяти. Как известно, один байт состоит из восьми бит, а каждый бит, в свою очередь, может принимать два значения: 1 или 0. Таким образом, набор из восьми бит может принять 2^8, то есть 256 возможных значений. Значит, если мы хотим хранить в

переменной типа *char* как отрицательные, так и положительные числа, в один байт у нас влезет диапазон значений от -128 до 127. Если же мы хотим хранить только положительные значения, у нас будет диапазон от 0 до 255. Как вы уже догадались, мы вплотную подошли к пониманию смысла спецификаторов *signed* и *unsigned*. Спецификатор *signed* указывает на то, что в переменной могут храниться как положительные, так и отрицательные значения, тогда как *unsigned* разрешает переменной принимать лишь положительные значения.

Особо любознательные найдут интересным тот факт, что в памяти значения signed и unsigned переменных одного типа имеют один и тот же вид. Например, содержимое байта памяти:

```
Номер бита →     7 6 5 4 3 2 1 0
Значение бита →  1 0 0 1 1 0 1 1
```

будет интерпретировано как число -101, если эта область памяти принадлежит переменной типа *signed char*, и как число 155 в случае *unsigned char*. По сути, спецификаторы *signed* и *unsigned* определяют интерпретацию последнего, восьмого бита. В случае signed-переменной «1» в этом бите означает отрицательное число, а «0» — положительное. Младшие 7 бит определяют значение. В случае же unsigned-переменной значение восьмого бита принимает полноправное участие в формировании значения байта.

Наверное, это место будет наиболее удачным для того, чтобы напомнить, как формируется десятичное значение из набора нулей и единиц. У каждой цифры («0» или «1») в двоичном числе есть определённый *вес*, который представляет собой возрастающие степени числа 2 и увеличивается с номером цифры (бита). Биты нумеруются справа налево.

Номер бита, n	→	7	6	5	4	3	2	1	0
Вес бита, 2^n →	128	64	32	16	8	4	2	1	
Значение бита	→	1	0	0	1	1	0	1	1

Чтобы получить десятичное значение, нужно сложить веса всех битов, имеющих значение «1». Найдём значение числа из примера, предположив, что оно беззнаковое. Как видно, в единицу выставлены биты 0, 1, 3, 4 и 7. Складываем их веса: 1 + 2 + 8 + 16 + 128 = 155. Возможно, это не самое распространённое описание систем счисления, однако, как показывает практика, оно более понятно, чем формальные тексты учебников.

Вернёмся теперь к спецификаторам типов. Мы уже выяснили, как работают спецификаторы *signed* и *unsigned*, разберёмся теперь с *long* и *short*. Эти два спецификатора влияют на размер памяти, выделяемой для переменной. Например, для *short int* будет выделено 2 байта, в то время как для *long int* — 4. Эти два спецификатора неприменимы к типу *char*. Для наглядности приведём табличку.

Тип данных	Размер памяти	Диапазон значений
signed char	1 байт	-128 – 127
unsigned char	1 байт	0 – 256
signed short int	2 байта	-32768 – 32767
unsigned short int	2 байта	0 – 65535
signed long int	4 байта	-2147483648 – 2147483647
unsigned long int	4 байта	0 – 4294967295

В данном курсе мы не будем использовать типы с плавающей точкой, поэтому они не вошли в таблицу.

По умолчанию переменные типа *char* имеют спецификатор *signed*, а переменные типа *int* — *signed* и *short*. Таким образом, эти записи будут эквивалентны:

```
signed int foo;     // explicitly signed and short by default
int bar;            // signed by default and short by default
short baz;          // signed by default and explicitly short
```

Ещё один важный аспект, касающийся переменных, и требующий внимания — это их инициализация. В простейшем случае, переменную можно объявить двумя способами:

```
int foo;
int bar = 123;
```

В первом случае происходит только выделение памяти для переменной *foo*, однако в эту память не записывается никакого значения. Это означает, что наша переменная будет содержать то значение, которое хранилось в этой области памяти до её объявления. При написании программы мы ни в коем случае не должны делать предположений о значениях неинициализированных переменных. При запуске программы переменная *foo*, скорее всего, будет содержать значение *0xFFFF*. Если переменная объявлена внутри функции, то есть вероятность, что при последующих вызовах для неё выделится та же область памяти, и тогда «новая» переменная будет иметь значение своей прошлой «инкарнации». Эта особенность отличает C от некоторых других языков, которые автоматически присваивают объявляемым переменным значение по умолчанию (например, 0 для численных типов).

Мы также можем явно присвоить значение переменной при её объявлении. Для этого нужно просто поставить знак равенства после её имени и написать нужное значение. Следующие две записи будут эквивалентны:

```
int bar = 123;

int bar;
bar = 123;
```

Последний вопрос, который мы должны обсудить перед получением зелёного пояса по переменным — это их область видимости. Рассмотрим ещё один пример.

```
int foo = 123;

int main() {
	int bar = 456;
	int result = foo + bar;
	int mainstuff = 911;

	return 0;
}

int notmain() {
	int bar = 789;
	int result = foo + bar;

	result = result + mainstuff; // Whooopsie.

	return 0;
}
```

Переменные могут быть двух типов в зависимости от их области видимости: **глобальные** и **локальные** (на самом деле, всё несколько сложнее, но пока этим не стоит забивать голову).

Глобальные переменные можно использовать в любой функции в пределах одного файла. Глобальные переменные объявляются в самом начале файла, перед всеми функциями. В примере, приведённом выше, *foo* — глобальная переменная.

Локальные переменные объявляются внутри блока, ограниченного фигурными скобками, и могут использоваться только внутри этого блока. В нашем примере внутри функции *main* объявлена переменная *bar* со значением 456. В переменную *result* записывается результат сложения глобальной переменной *foo* и локальной переменной *bar*. Для функции *main* значение *result* будет равно 579.

Функция *notmain* очень похожа на функцию *main*: она тоже объявляет переменную *bar*, и тоже складывает её значение с глобальной переменной. Однако, в отличие от *main*, значение переменной *bar* в *notmain* равно 789, а значит результат будет равен 912. Как видите, обе функции используют одно значение глобальной переменной *foo* и разные значения собственных локальных переменных *bar*.

Студент, писавший функцию *notmain*, прогуливал лекции по С, поэтому решил использовать в ней переменную *mainstuff*, локально объявленную в *main*. Логично предположить, что его программа не скомпилируется, и он будет ходить на пересдачу до тех пор, пока более опытные товарищи не расскажут ему про область видимости переменных.

На этой оптимистичной ноте мы и закончим обзор переменных, а в следующем разделе рассмотрим операции, которые с ними можно производить.

Операции над переменными

Над переменными можно проводить как тривиальные арифметические операции (как в первом примере прошлого раздела), так и более сложные, логические. Однако прежде, чем мы начнём говорить об операциях, стоит обсудить такое, на первый взгляд, простое действие как присваивание.

Очевидно, что переменной можно присвоить значение, написав после неё знак «равно» и подставив нужное значение. Но что будет, если написать следующий код?

```
unsigned int foo = 65535;
unsigned char bar = foo;
```

Мы объявили переменную *foo* типа *int*, присвоили ей значение, а потом присвоили значение переменной *foo* переменной *bar*, имеющей тип *char*. Но разве так можно делать? Ведь переменная типа *int* занимает два байта, тогда как переменная типа *char* — всего один! Оказывается, так делать можно, и есть одно-единственное правило, которое действует во всех подобных случаях. Оно заключается в том, что присваиваемое значение приводится к типу переменной, стоящей слева от знака «равно». В нашем случае от значения *foo* будет взято только 8 младших бит и они будут присвоены *bar*. Таким образом, переменная *bar* после присваивания будет иметь значение 255.

```
Номер бита →      15  14  13  12  11  10  9   8     7  6  5  4  3  2  1  0
Значение бита →   1   1   1   1   1   1   1   1     1  1  1  1  1  1  1  1
                    ------------------------------    -----------------------
                     Эта часть отбрасывается          А эта присваивается bar
```

Ещё один вопрос, который стоит обсудить, пока мы говорим о присваивании — это форма записи числовых констант. При использовании компилятора AVR GCC можно записывать числа тремя способами: в десятичной системе (как мы до сих пор делали во всех примерах), в двоичной и шестнадцатеричной системах.

Запись чисел в двоичной системе особенно удобна при программировании микроконтроллеров, где часто приходится задавать значения отдельных битов в регистрах, отвечающих за те или иные настройки. Чтобы записать число в двоичной системе, перед ним нужно поставить префикс *0b*.

```
unsigned char foo = 123;
unsigned char bar = 0b01111011;
unsigned char baz = 0x7b;
```

В этом примере все три переменные имеют одинаковые значения. При присвоении значения переменной *bar* мы использовали двоичную форму записи, которая позволила нам наглядно установить значение каждого бита. Для переменной *baz* мы использовали шестнадцатеричную форму записи с префиксом *0x*. Прочитать больше про шестнадцатеричную систему счисления можно в Википедии.

Арифметические операции

Теперь приведём сводную таблицу простейших арифметических операторов.

Оператор	**Описание**
+	Сложение
-	Вычитание
*	Умножение
/	Деление
%	Остаток от деления
++	Инкремент
--	Декремент

Большинство операций изучались ещё в первом классе, но на некоторых всё же стоит остановиться подробнее.

Например, нужно помнить, что использование целочисленных переменных (таких, как *int*) для хранения результатов математических операций приведёт к отбрасыванию дробной части при делении. В следующем примере переменная *foo* будет иметь значение 3, а не 3,3333(3).

```
int foo = 10.0 / 3;
```

Чтобы получить остаток от деления числа x на число y, можно использовать оператор %. В этом примере значение переменной *foo* будет равно 1.

```
inf foo = 10 % 3;
```

Две операции, присущие большей частью языкам программирования — это инкремент и декремент. Эти операции позволяют увеличить или уменьшить значение переменной на единицу. Следующие записи будут эквивалентны:

```
foo = foo – 1;      // substracts 1 from foo
foo--;       // does exactly the same
```

С инкрементом и декрементом также связан один нюанс, который часто приводит к трудноуловимым багам в программах. Дело в том, что эти два оператора могут располагаться как перед именем переменной, так и после него. Такие формы записи называются префиксной и постфиксной. Они не имеют значения, если вы просто увеличиваете или уменьшаете значение переменной. Однако форма оператора инкремента или декремента имеет значение, если вы сразу присваиваете результат операции другой переменной.

```
int foo = 13;
int bar = foo++;
int baz = ++foo;
```

На второй строке мы сначала присваиваем *bar* текущее значение *foo*, и лишь затем увеличиваем его на единицу. Таким образом, после выполнения второй строки *bar* будет иметь значение 13, а *foo* — 14. На третьей строке мы используем префиксную форму записи, а значит, сначала увеличиваем значение *foo*, и только после этого присваиваем его *baz*. После выполнения третьей строки обе переменные, *baz* и *foo*, будут иметь значение 15.

Вопрос на зачёт: какое значение примет bar?

```
int foo = 5;
int bar = ++foo + ++foo;
```

При применении арифметических операций нужно также помнить о переполнении. Например, после выполнения этого кода переменная *foo* будет иметь значение 0, так как верхняя граница для типа *unsigned int* равна 65535.

```
unsigned int foo = 65535;
foo++;
```

Кроме обычных арифметических операторов в языке С существует набор сокращённых операторов. Они используются в тех случаях, когда нужно произвести какое-то действие с текущим значением переменной и сохранить результат в эту же переменную. Например, следующие две записи будут эквивалентны.

```
foo = foo + 5;      // Add 5 to foo
foo += 5;           // This adds 5 to foo as well
```

Любой оператор, который можно применить к двум операндам, можно также записать и в сокращённой форме.

```
foo += 5;
bar -= 10;
baz /= 2;
foo *= 3;
bar &= 1;
baz |= 1;
```

Операции сравнения

В языке C есть несколько классов операций, результатом которых становится булевское значение (истина/ложь*; true/false). Стандарт C89 определяет значение *false* как ноль, и значение *true* как любое другое число.

```
int foo = 0;        // equals FALSE
int bar = 1; // equals TRUE
int baz = -32;      // equals TRUE as well
```

Операциями, возвращающими значения *true* и *false*, являются, например, операции сравнения.

Оператор	**Описание**
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно
==	Равно
!=	Не равно

Смысл большей части перечисленных операторов очевиден, особое внимание стоит уделить лишь оператору «равно». Всегда нужно помнить, что для присвоения значения переменной используется «одинарное равно», а для сравнения двух переменных — «двойное». Дело усугубляется тем, что компилятор не выдаст ошибки при неправильном использовании оператора сравнения.

```
int foo = 0;

if (foo = 5) {
      // do the course project
} else {
      // have a beer
}
```

Приведённый код скомпилируется, однако, будет работать совсем не так, как планировал автор. Вместо того чтобы сравнить значение *foo* (которое изначально равно 0,

то есть *false*) с числом 5, компилятор сначала присвоит *foo* значение 5, а потом обработает инструкцию *if*. Так как 5 не равно 0, всё выражение будет трактовано как *true*. Правильный код будет выглядеть так:

```
int foo = 0;

if (foo == 5) {
	// do the course project
} else {
	// have a beer
}
```

В остальном операторы сравнения работают очень просто: выдают *true*, если выражение соответствует истине, и *false* — если нет.

Логические операции

Второй класс операций, которые возвращают булевские значения — это логические операции. Существует три основные логические операции, поддерживаемые языком С.

Оператор	**Описание**
&&	Логическое «И» (AND)
||	Логическое «ИЛИ» (OR)
!	Логическое «НЕ» (NOT, отрицание)

Каждый из этих операторов может применяться к булевским значениям (а учитывая «лояльное» определение стандартом булевских значений, вообще ко всем переменным). Оператор *&&* возвращает *true* только в том случае, если оба операнда равны *true*. Оператор *||* возвращает *true* если хотя бы один из операндов имеет значение *true*. Оператор *!* просто возвращает противоположное значение операнда.

Приведём стандартную таблицу истинности.

foo	**bar**	**foo && bar**	**foo || bar**	**!foo**
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

С помощью логических операторов можно делать сложные проверки, например:

```
if (!foo && (bar || baz))
	DestroyTheWorld();
```

В данном случае также иллюстрируется обработка скобок в С. Если в некотором выражении присутствуют скобки, компилятор обрабатывает их на основе правил обычной математики. Поэтому, если вы хотите написать сложное условие, и не знаете как, просто воспользуйтесь здравым смыслом и знаниями, полученными в школе — этого будет вполне достаточно.

Побитовые операции

Мы подошли к изучению самого хитрого класса операторов — побитовых. Эти операторы манипулируют не с целыми значениями переменных, а с их элементарными частями — битами. Понимание этих операций очень важно при программировании микроконтроллеров, поэтому рассмотрим их подробнее.

Побитовые операции очень похожи на логические, однако, они возвращают не булевские значения, а значение переменной после проведения над ней операции.

Оператор	Описание
&	Побитовое «И» (AND)
|	Побитовое «ИЛИ» (OR)
~	Побитовое «НЕ» (NOT)
^	Исключающее «ИЛИ» (XOR)
<<	Побитовый сдвиг влево
>>	Побитовый сдвиг вправо

Побитовые операции работают следующим образом: берутся два операнда, после чего к **каждому** из битов применяется указанная операция. Рассмотрим пример.

```
        11010010
&       10110111
        --------
        10010010
```

В этом примере мы взяли два двоичных числа и применили к ним побитовое «И». Для каждой пары бит в этих числах мы применили логическую операцию «И», как если бы эти биты были булевскими переменными. Из результирующих битов (значений *true* и *false*, получившихся после каждой логической операции) и складывается ответ. Приведём ещё один пример.

```
        11010010
|       10110111
        --------
        11110111
```

А вот пример использования побитовой операции в коде программы:

```
unsigned char foo = 0b01110111;
if (foo & 0b01000000)
{
        // do stuff
}
```

Этот пример иллюстрирует очень распространённую технику проверки значения определённого бита, называемую *маскированием*. Допустим, нам нужно узнать, имеет ли бит номер 6 значение «1», и предпринять какие-то действия, если это так. Мы применяем побитовую операцию «И» к интересующей нас переменной и константе, называемой «маской». В маске в «1» выставлены те биты, значение которых нас интересует в

переменной *foo* (обычно в «1» выставляется только один бит маски, иначе результат проверки будет неоднозначным). В нашем случае операция будет иметь такой результат:

```
foo         →     01110111
& маска     →     01000000
                  --------
результат   →     01000000
```

Как видите, в результате мы получим десятичное значение 128, которое превратится в *true* при проверке. Рассмотрим случай, когда переменная *foo* будет иметь другое значение:

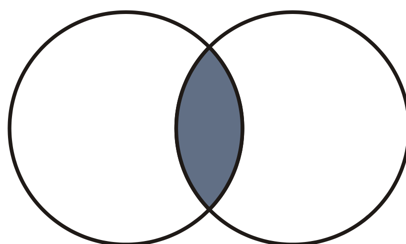
```
foo         →     10110111
& маска     →     01000000
                  --------
результат   →     00000000
```

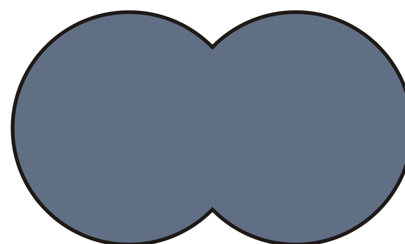
В этом случае бит, проверяемый маской, равен «0», а значит и весь результат будет равен 0, что эквивалентно *false*.

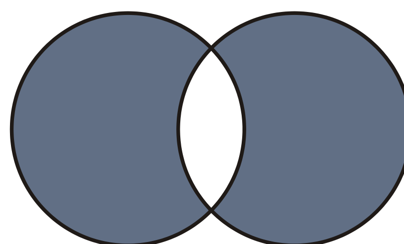
Таблицы истинности логических операторов OR, AND и NOT соответствуют таблицам истинности соответствующих побитовых операций. Также существует одна хитрая побитовая операция XOR, таблица истинности которой приведена ниже:

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

Для наглядности приведём графическое представление некоторых операций.







Операции сдвига

Осталось рассказать лишь про побитовый сдвиг. Это очень простая, и в то же время мощная операция, суть которой заключается в том, что все биты переменной сдвигаются в ту или иную сторону. Лучше всего эта операция объясняется на примерах.

```
      11001001                             11001001
<< 1                                 >> 1
      10010010                             01100100
<< 2                                 >> 2
      00100100                             00110010
<< 3                                 >> 3
      01001000                             00011001
<< 4                                 >> 4
      10010000                             00001100
```

Как видно из примера, левым операндом является некое значение, а правым — количество бит, на которое нужно произвести сдвиг. Биты сдвигаются в указанную сторону на указанное количество позиций, при этом, биты, вышедшие «за границу», затираются, а на «освободившиеся» места подставляются нули. Побитовый сдвиг имеет также и «физический смысл»: сдвиг влево эквивалентен умножению на 2, а сдвиг вправо — делению на 2 с округлением.

Массивы и указатели

В этом разделе мы рассмотрим одну из самых сложных тем в С — указатели. Есть только одна вещь, которую сделать сложнее, чем понять указатели — это объяснить указатели.

Массивы

Для разминки начнём с более простой темы и разберёмся, что же собой представляют массивы. Массив — это набор переменных одного типа, объединённых под одним именем. Допустим, что каждые пять секунд нам нужно получать некое значение (например, с АЦП) и сохранять его в соответствующую переменную. Решением «в лоб» будет объявить пять переменных и работать с ними:

```
int val1, val2, val3, val4, val5;

if (curSecond == 0)
      val1 = GetADC();
else if (curSecond == 1)
      val2 = GetADC();
// And so on and so far
```

Будем считать, что в переменной *curSecond* за нас кто-то обновляет значение текущей секунды, а функция *GetADC()* получает данные с АЦП. Как видите, здесь мы записываем по одному условию для каждого значения текущей секунды: 0, 1, 2 и т. д.

Однако это не самое удачное решение. Гораздо проще было бы объявить одну большую переменную, которая состояла бы из пяти маленьких частей, верно? Вот именно для этого и существуют массивы. Например, предыдущий пример можно сократить всего до двух строк.

```
int values[5];

values[curSecond] = GetADC();
```

Разберёмся, что же здесь произошло. Сначала мы объявили массив *values*, состоящий из пяти элементов типа *int*. В общем случае массивы объявляются следующим образом:

```
тип имя_массива[количество_элементов];
```

На второй строке происходит присваивание значения АЦП элементу массива, номер которого равен текущему значению *curSecond*. Рассмотрим ещё один пример.

```
int foo[3];
int bar[3] = {123, 456, 789};
int baz;

foo[0] = 123;
foo[1] = 456;
foo[2] = 789;

baz = foo[0];       // baz == 123
baz = bar[1];       // baz == 456
baz = bar[3];       // Index out of bounds, access violation,
                    // nuclear holocaust and bad karma.
```

На первой строке мы объявили массив *foo* из трёх элементов типа *int* и не инициализировали его. Так же, как и в случае неинициализированных переменных, значения элементов неинициализированного массива являются случайными.

На второй строке мы объявили такой же массив, и сразу инициализировали его. Как видите, инициализация делается с помощью фигурных скобок, в которых через запятую перечисляются значения элементов массива.

Далее идёт блок, в котором мы по очереди присваиваем значения элементам массива *foo*. В этом месте нужно сделать одно очень важное замечание: **индексация массивов в С производится с нуля, и никак иначе!** Никаких паскалеподобных компромиссов с явным указанием основания для индексации, которые только запутывают нормальных людей. Итак, ещё раз.

1. При объявлении массива в квадратных скобках указывается **количество** элементов. Если хотите объявить массив из трёх элементов, в квадратных скобках будет значение 3.
2. Доступ к элементам массива в программе производится по номерам, начинающимся с **нуля**. Если вы объявили массив из трёх элементов, то номером последнего его элемента будет **2**.

Как вы уже догадались, последняя строка неверна, так наш массив состоит из трёх элементов, а мы обратились к четвёртому. Нужно очень внимательно следить за индексацией массивов, так как компилятор не производит никакой проверки на выход индекса за границы. Приведенный выше код скомпилируется без единого предупреждения, однако, при попытке выполнить последнюю инструкцию произойдёт нарушение доступа к памяти. Вы не захотите вылавливать подобные баги в прошивке микроконтроллера, уж поверьте.

Как уже было показано в одном из примеров, для доступа к элементам массива можно использовать не только явно заданные значения, но и значения других переменных (в 90% случаев массивы используются именно по такому сценарию).

```
int foo[5] = {1, 2, 3, 4, 5};
int bar = 2;
int baz;

baz = foo[2];
baz = foo[bar];
```

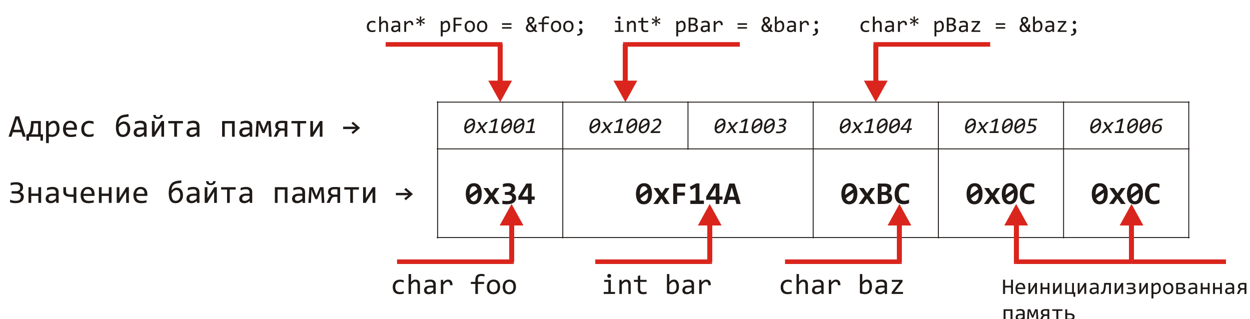
В этом примере мы объявили массив и присвоили переменной *baz* значение одного из его элементов двумя способами. Очевидно, что последние две строки дают одинаковый результат.

Указатели

В самом первом разделе, посвящённом переменным, мы рассказали вам не всю правду. Мы сказали, что при объявлении переменных для них выделяются области памяти, однако, умолчали о том, как к ним происходит доступ на самом деле. Пришло время исправить эту ситуацию.

В памяти нет никакой информации об именах переменных, равно как нет её и в скомпилированной программе. Машинные инструкции ничего не знают об именах, которые вы ввели для своего удобства. Эти инструкции обращаются к ячейкам памяти по их *адресам*. В реальной жизни, чтобы куда-нибудь попасть, вам нужно знать адрес места, в которое вы хотите попасть. Так же происходит и с машинными инструкциями. Чтобы получить или изменить значение переменной, они используют их адреса.

Каждая ячейка памяти (читай: каждый байт) имеет свой уникальный *адрес*, по которому к этой ячейке можно получить доступ. Адрес — это самое обычное число, которое обычно записывается в шестнадцатеричной форме.



На этом рисунке схематично представлен блок памяти, разделённый на элементарные ячейки (байты). Когда мы объявляем переменные обычным способом, происходит выделение необходимой памяти и переменные содержат значения, хранящиеся в выделенных ячейках. На самом деле, обращение к этим ячейкам происходит по их адресам, но эти подробности от нас скрыты.

Однако бывают случаи, когда возникает необходимость работать непосредственно с адресами ячеек памяти. Именно для этого и служат указатели. *Указатель* — это переменная, которая содержит не значение ячейки памяти, а её *адрес*. Рассмотрим пример.

```
int  foo = 5;
int* bar = &foo;

foo = 6;     // *bar == 6, because
             // bar holds foo’s address

*bar = 7;    // foo == 7 now, but bar remains the same
             // because foo’s address didn’t change
```

На первой строке мы объявили обычную переменную *foo* и присвоили ей значение 5. На второй строке мы объявили указатель на переменную типа *int* и присвоили ему адрес переменной *foo*.

Как видите, переменная-указатель объявляется как обычная переменная с добавлением звездочки (*) после типа переменной. Здесь стоит обратить внимание на

небольшой нюанс. Дело в том, что C очень лояльно относится к пробелам, символам табуляции и перевода строки. Поэтому указатели можно объявлять по-разному:

```
int* foo;           // foo is a pointer
int *bar;           // bar is also a pointer
int * baz;          // Even baz is a pointer
int* a, b, c;       // Only a is a pointer here!
```

Звёздочку можно ставить как возле типа переменной, так и возле имени переменной. Использование конкретного стиля — личное дело каждого программиста. Однако нужно помнить, что при объявлении нескольких указателей на одной строке (строка 4 в примере) звёздочку надо ставить возле имени **каждой** переменной, которую вы хотите сделать указателем. В приведённом примере указателем будет только переменная *a*, в то время как *b* и *c* будут иметь обычный тип *int*. Из-за такой путаницы указатели рекомендуется объявлять отдельно, по одному на строке.

Вернёмся теперь к первому примеру. Мы уже выяснили, что переменная *bar* является указателем типа *int*, а значит, может содержать адрес любой переменной типа *int*. На самом деле, в указателях любого типа хранятся адреса одинакового размера, тип указателя играет роль лишь в интерпретации значения, получаемого по этому адресу. Например, в указателе типа *char** хранится адрес байта памяти. Когда мы захотим получить значение ячейки, на которую указывает этот адрес, компилятор автоматически сгенерирует код для получения ровно одного байта по указанному адресу (так как тип *char* занимает один байт). Если же у нас есть указатель типа *int**, в нём будет храниться всё тот же адрес одного байта памяти, однако, при получении значения по этому адресу будут считаны два байта (так как тип *int* занимает 2 байта).

Теперь рассмотрим подробнее способы работы с указателями. Существуют два основных действия: *взятие адреса переменной* и *разыменовывание указателя*. Первое позволяет получить адрес ячейки памяти, в котором хранится значение переменной, а второе — изменить значение этой ячейки при известном адресе. В нашем примере при объявлении указателя *bar* мы сразу инициализировали его адресом переменной *foo*. Как вы уже догадались, для получения адреса переменной используется оператор “&”.

```
тип_указателя* имя_указателя = &имя_переменной;
```

Оператор “&” можно поставить перед именем любой переменной, и результатом этого действия будет адрес первой из ячеек памяти, которые занимает данная переменная.

Для доступа к значению памяти, адресуемой указателем, существует оператор разыменовывания “*”. Для того, чтобы получить или изменить значение памяти, на которую ссылается указатель, достаточно поставить звёздочку перед именем этого указателя.

```
*имя_указателя = значение_переменной;
```

Символ “*” довольно популярен в C: он используется для объявления указателя, и он же применяется для его разыменовывания.

Имея такой багаж знаний, ещё раз посмотрим на первый пример с указателями. Очень важно понимать, что после того, как мы присвоили указателю *bar* адрес переменной *foo*, *bar* содержит именно **указатель** на значение, а не само значение. Таким

образом, если мы изменим значение переменной *foo*, указатель *bar* не изменится, так как переменная *foo* осталась на своём месте в памяти, мы просто изменили *значение* этой памяти. Если же мы разыменуем указатель *bar* с помощью оператора “*”, то мы, по сути, будем присваивать значение переменной *foo*, так как именно на него указывает *bar*. Именно поэтому после выполнения инструкции “*bar = 7;” переменная *foo* тоже примет значение 7.

Связь между указателями и массивами

Мы не случайно объединили темы указателей и массивов в один раздел. Между ними существует тесная связь, о которой мы сейчас и поговорим. До сих пор использование нами массивов сводилось к доступу к их элементам по индексу.

```
int foo[3];

foo[0] = 1;
foo[1] = 2;
foo[2] = 3;
```

Но что представляет собой сама переменная *foo*, если записать её отдельно, без квадратных скобок? А представляет она собой не что иное, как указатель, в котором хранится адрес первого элемента массива. Зная эту тайну, мы можем написать следующий код.

```
int foo[3] = {1, 2 ,3};
int* bar;

bar = foo;          // No “&” operator here!
bar[0] = 123;       // Now we can use bar like it was
                    // an array from the start!
```

Здесь мы объявили массив *foo* типа *int* и указатель *bar* того же типа. Зная, что *foo* — это такой же указатель, мы можем спокойно присвоить его значение указателю *bar*. После чего мы можем использовать указатель *bar* с квадратными скобками для доступа к элементам массива *foo*. Да, всё верно, на низком уровне переменные *foo* и *bar* представляют собой одно и то же за тем лишь исключением, что указателю *bar* вы можете присвоить адрес любой переменной, а *foo* всегда будет указывать на первый элемент своего массива. Рассмотрим ещё один комплексный пример.

```
int foo[3] = {1, 2, 3};
int* bar;
int baz;

bar = foo;          // can me haz array?

*bar = 123;         // We changed the first element of the array
                    // foo[0] == 123 now

bar[1] = 456;       // foo[1] == 456

*(bar + 2) = 789;   // Hardcore pointer arithmetic,
                    // foo[2] == 789

baz = bar[2];       // This one is simple, baz == bar[2] == foo[2] == 789;

bar = &foo[1];      // bar now points to the second element of foo

bar[1] = 0;         // foo[3] == 0. Yes, that’s right.
                    // If you understood this, you’ll probably
                    // make it till the end of the semester ;)
```

Некоторые моменты здесь требуют объяснения. Например, в инструкции

```
*(bar + 2) = 789;
```

мы использовали так называемую указательную арифметику. Кроме обычного присваивания указателей с ними можно проводить операции сложения, вычитания, инкремента и декремента (и никаких других). При проведении этих операций адрес, хранящийся в указателе, изменяется не на то количество байт, которое указано после знака «+» или «-», а на размер, занимаемый типом данных, умноженный на число после знака. Например, если у нас есть указатель типа *int* foo*, в котором хранится адрес 0x1001, то после выполнения инструкции "foo + 2" *foo* будет хранить не 0x1003, а 0x1005, так как тип *int*, с которым объявлен указатель, занимает два байта. Таким образом, в нашем примере мы прибавили к указателю первого элемента значение 2, перейдя, таким образом, к третьему элементу, а затем разыменовали указатель, присвоив значение 789 третьему элементу массива.

Инструкция

```
bar = &foo[1];
```

демонстрирует нам, что квадратные скобки, по сути, действуют как оператор разыменования. Здесь мы сначала получили значение первого элемента массива, а потом снова взяли его адрес и присвоили его указателю *bar*. После этого получилось, что *bar* указывает уже не на первый, а на второй элемент массива *foo*, поэтому и конструкция "bar[1]" указывает на третий элемент вместо второго.

Тех, кто ничего не понял из написанного на этой странице, можно успокоить: это довольно сложный материал, и он не входит в необходимый минимум знаний. Однако если вы с ним разберётесь, то получите мощный инструмент, который пригодится в том случае, если вам когда-нибудь доведётся столкнуться с низкоуровневым программированием.

Управляющие конструкции

В этом разделе мы рассмотрим специальные конструкции языка С, которые позволяют производить различные проверки, организовывать циклы и ветвления в логике программы.

Оператор "if"

Начнём с самого простого и наиболее часто используемого оператора. Оператор *if* позволяет провести проверку некоторых условий и выполнить определённые действия на основании результатов этой проверки. Сразу рассмотрим пример.

```
int foo = 5;
if (foo > 10)
{
	// do stuff
}
else
{
	// do other stuff
}
```

Как видите, после оператора *if* в круглых скобках следует проверяемое условие. Если результат проверки будет равен *true*, будет выполнен блок, идущий сразу после оператора *if* и выделенный фигурными скобками. Также после этого блока может быть размещено ключевое слово *else* и ещё один блок кода, который будет выполнен в том случае, если результат проверки окажется равным *false*. Блок *else* не является обязательным: если его нет и результат проверки равен *false*, блок, идущий после оператора *if,* просто не будет выполнен.

Если блок кода, который необходимо выполнить, состоит только из одной инструкции, фигурные скобки можно опустить и записать условие в виде:

```
if (something)
	foo();
else
	bar();
```

Если условие, которое нужно проверить, нельзя уместить в обычную бинарную логику *true/false*, можно объединить подряд несколько операторов *if*:

```
int D = b * b – 4 * a * c;
int x1, x2;

if (D < 0)
	// Complex numbers
else if (D == 0)
	x1 = x2 = -b / (2 * a);
else
{
	x1 = (-b + sqrt(D)) / (2 * a);
	x2 = (-b - sqrt(D)) / (2 * a);
}
```

Оператор “switch”

Бывают случаи, когда нужно сделать много однотипных проверок. Для этого можно использовать несколько операторов *if*, однако, эффективнее будет применить *switch*. Рассмотрим пример.

```
int foo = 5;

switch (foo)
{
	case 0:
		ZeroFunc();
		break;
	case 1:
		OneFunc();
		break;
	case 2:
		TwoFunc();
		break;
	default:
		DefaultFunc();
}
```

Здесь после оператора *switch* в скобках находится проверяемое выражение, а внутри блока *switch* расположены возможные результаты этой проверки. Каждый из

результатов обозначается ключевым словом *case* и двоеточием. После двоеточия может следовать сколько угодно инструкций, которые обязательно должны заканчиваться инструкцией *break*. Если не поставить *break*, будут выполнены все остальные инструкции из последующих блоков *case*. Инструкции из блока *default* выполняются в том случае, если ни один из блоков *case* не удовлетворил результату проверки.

Оператор “for”

Оператор *for* позволяет организовывать циклы. *Цикл* — это последовательность действий, выполняемых много раз с незначительными изменениями или над разными объектами. Как водится, начнём с примера.

```
int foo[6] = {4, 8, 15, 16, 23, 42};
int bar = 0;
int i;

for (i = 0; i < 5; ++i)
{
      bar += foo[i];
}
```

Сначала мы объявляем и инициализируем массив из шести элементов, затем объявляем переменную для хранения результата и *переменную-счётчик*. Счётчик используется для того, чтобы хранить количество выполненных итераций цикла.

Далее идёт сам цикл. Оператор *for* имеет следующую форму:

```
for (инициализация_счётчика; условие_выполнения; выражение)
{
}
```

Как видите, блок, заключённый в скобки, имеет три части, разделённые точками с запятыми. Каждая из этих частей может представлять собой абсолютно любое выражение. Первая часть выполняется один раз перед входом в цикл. Обычно её используют для присвоения начального значения переменной-счётчику (в нашем примере *i = 0;*). Вторая часть представляет собой условие выполнения цикла. Это условие работает как оператор *if*, который выполняется перед каждой новой итерацией. Если результат проверки равен *true*, итерация выполняется; если *false* — цикл завершается. В нашем примере мы проверяем значение счётчика *i < 5*. Третий блок выполняется *после* каждой итерации. Обычно этот блок используется для увеличения значения счётчика (в нашем примере *i++*).

Код, размещённый в фигурных скобках за оператором *for*, называется *телом цикла* и выполняется до тех пор, пока не будет произведён выход из цикла. Кстати, выход из цикла можно сделать и вручную, не дожидаясь выполнения условия, указанного в операторе *for*. Для этого в теле цикла нужно поместить оператор *break*. Например:

```
for (i = 0; i < 10; i++)
{
      if (red_alert)
            break;
      // Do normal stuff in a loop
}
```

Циклы часто используются для обработки массивов, при этом счётчик цикла используется как индекс элемента массива. Например, цикл в первом куске кода вычисляет сумму элементов массива *foo*, которая будет находиться в *bar* после завершения цикла.

Оператор “while”

Оператор *while* предоставляет нам ещё один способ организовать циклы. Он устроен проще, чем оператор *for* и может применяться в тех случаях, когда цикл не нуждается в счётчиках.

```
while (!SkyIsBlue())
{
	RemoveClouds();
}
```

Как можно понять из этого примера, цикл *while* выполняется до тех пор, пока выражение в скобках равно *true*.

Цикл *while* очень важен при программировании микроконтроллеров, так как именно с его помощью организуется главный цикл (подробнее о главном цикле — в первой лабораторной):

```
while (1)
{
	CheckButtons();
	GeneratePWM();
	GetADC();
	wdt_reset();
}
```

Функции

Функции — это очень важная часть любой программы на Си. С их помощью можно структурировать программу, выделив определённые блоки кода и обращаясь к ним по мере необходимости.

Функцией называется блок кода, имеющий имя и, возможно, возвращающий какое-то значение.

```
int add(int a, int b);

int main()
{
	int foo = add(5, 2);
	int bar = add(6, 8);

return 0;
}

int add(int a, int b)
{
	return a + b;
}
```

Разберёмся, что же здесь произошло. На первой строке, в области глобальных переменных, мы объявили *сигнатуру* функции. Сигнатура состоит из типа возвращаемого функцией значения, имени функции и списка параметров. Сигнатура функции объявляется для того, чтобы все функции в пределах одного файла знали о существовании друг друга. Для функции *main* сигнатуру можно не объявлять, так как она вызывается автоматически при запуске программы и вызывать её вручную крайне не рекомендуется (если очень хочется, то можно, однако, это сильно смахивает на индусский код).

После функции *main* следует *реализация* функции *add*, сигнатуру которой мы объявили ранее. Как видите, мы передаём в функцию две переменные типа *int*. Они называются *параметрами* или *аргументами* функции. Существует несколько способов передачи параметров. В нашем примере используется передача по значению. Это означает, что при входе в функцию создаются новые переменные с указанными именами, которым присваиваются переданные значения. По сути, параметры функции эквивалентны локальным переменным, объявленным в теле этой функции. Мы можем использовать их как обычные переменные, и даже изменять их значения. Хоть это и никак не отразится на значениях, переданных вызывающей функцией, изменять аргументы не рекомендуется хотя бы только потому, что это ухудшает читаемость кода и затрудняет его понимание.

Основная задача многих функций — обработать некую информацию, принятую в виде аргументов и вернуть результат. Функция может вернуть одно значение любого типа. Этот тип указывается в сигнатуре функции перед её именем. Возвращаемое значение — это ещё одна переменная, которую можно использовать наравне с остальными. Как видите, в функции *main* мы присваиваем результат выполнения функции *add* обычной переменной. Для возврата значения из функции служит ключевое слово *return*. Как только процесс выполнения программы подходит к этому оператору, производится выход из текущей функции и возврат в вызывающую функцию. В нашем примере происходит такая последовательность действий:

```
Объявление переменной foo →
вызов функции add с параметрами 5 и 2 →
вход в функцию add →
сложение аргументов →
return →
возврат в функцию main →
присваивание результата выполнения add переменной foo →
продолжение выполнения функции main.
```

Ключевое слово *return* можно использовать сколько угодно раз. Важно лишь помнить, что после возврата выполнение функции немедленно прекращается. Например, в следующем коде третий оператор *return* никогда не будет выполняться:

```
int foo(int bar)
{
	if (bar < 0)
		return 1;
	else
		return 2;

	return 3; // Will never get here.
}
```

Функция может и не возвращать никакого значения. В этом случае она объявляется с типом возврата *void* и не требует обязательного оператора *return* в конце.

```
void foo(int bar)
{
	if (bar > 0) {
		DoGreaterStuff();
		return;
	}

	DoLesserStuff();
} // We don’t need to return anything from void function
```

Стоит ещё раз повторить, что возврат в вызывающую функцию после завершения вызываемой происходит автоматически. Это значит, что если мы вызываем функцию *add* из функции *main*, нам не нужно ещё раз вызывать *main* в самом конце *add*. После завершения функции *add* выполнение программы продолжится с того самого места *main*, на котором оно прервалось.

Теперь рассмотрим ещё один способ передачи параметров — по указателю. В этом случае в функцию передаётся не значение переменной, а указатель на неё. Таким образом, внутри вызываемой функции мы можем изменить значение переменной, указатель на которую нам передали.

```
void foo(int* arg);

int main()
{
	int bar = 5;
	foo(&bar);
	// bar == 6 now.
	return 0;
}

void foo(int* arg)
{
	(*arg)++;
}
```

Мы объявили функцию как и раньше, однако, теперь передаём туда не значение переменной типа *int*, а указатель на эту переменную. Это даёт нам возможность разыменовать указатель в теле функции и получить доступ к исходному значению переменной. Есть несколько сценариев использования передачи параметров по указателю. Один из них — передача массивов. Массивы в C не могут быть переданы по значению. Чтобы передать массив в функцию, нужно использовать указатель.

```
void foo(int* arg);

int main() {
	int bar[3] = {1, 2, 3};
	foo(bar);
}

void foo(int* arg) {
	arg[0] = arg[1] + arg[2];
}
```

Мы воспользовались тем, что переменная массива является указателем на первый его элемент и передали этот массив по указателю в функцию *foo*.

Передачу аргументов по указателю можно также использовать в тех случаях, когда функция должна вернуть больше одного значения. Одно из значений можно возвращать с помощью оператора *return*, а для всех остальных объявить переменные в вызывающей функции и передать их по указателю в вызываемую функцию.

Символы и строки

Важной частью любой программы является вывод некоей информации. Это могут быть результаты расчётов, отчёт о текущем состоянии или просто отладочная информация. Обычно информация выводится в виде простого текста, но как этот текст представлен в исходной программе? В этом разделе мы попробуем разобраться в этом вопросе.

Символы

В стародавние времена, когда компьютеры были большими, а программистов было мало, основным «человеческим» языком, на котором говорили компьютеры, был английский. Чтобы выводить на экран или принтер какую-либо информацию, создатели компьютеров решили договориться о способе сопоставления числовых значений и печатных символов, которые бы им соответствовали (это и логично, так как компьютер знает только про числа, а не про какие-то там символы или строки). Так родился стандарт ASCII, который устанавливал соответствие между числами от *32* до *127* буквам латинского алфавита и некоторым служебным символам. Числа от 0 до 31 считались служебными и на экран не выводились, а числа от 128 до 255 резервировались для специальных символов. Таким образом, для хранения в памяти одного символа было достаточно одного байта, который, как известно, может принимать значения от 0 до 255. Именно поэтому однобайтовый тип переменных в C и называется *char* — от английского “character”. Всё это означает, что символы хранятся в памяти как обычные числа, и их интерпретация зависит только от того, как их применять. Вы вполне можете проводить небольшие вычисления с помощью типа *char*, и, в то же время, использовать этот тип как репрезентацию символа.

```
char foo = 32;
char bar = foo + 2;
```

В переменной *bar*, как вы уже поняли, будет находиться число 34. Вы можете использовать его как обычное число в своём коде. Однако если вы передадите это число в какую-нибудь функцию, которая выводит символы на экран, она интерпретирует число 34 как символ “C”. Вам необязательно знать числа, которые соответствуют символам. Можно написать, например, такой код:

```
char foo = 'A';
char bar = foo + 2;
```

Компилятор автоматически интерпретирует символ 'A' в значение *32*, поэтому этот отрывок кода будет равнозначен предыдущему.

После принятия стандарта ANSI все жили долго и счастливо, пока в один прекрасный день внезапно не выяснили, что далеко не все люди на планете говорят по-английски. Вторым неожиданным открытием был тот факт, что верхних 127 символов,

зарезервированных стандартом как специальные, не хватит для того, чтобы представить все символы из других языков. Немного поразмыслив, программисты решили пойти на компромисс и создали концепцию кодовых страниц. Фокус заключался в том, что каждый язык должен сам определять, какие числа из диапазона 127-255 соответствуют каким символам из этого языка. Это значит, что значение 150 могло означать один символ в русской кодировке и совсем другой символ в японской. В памяти символы всё так же продолжали храниться в виде чисел от 0 до 255, однако изменился способ, с помощью которого интерпретировались эти числа. Теперь для отображения символов какого-то специфического языка нужно было указывать соответствующую кодовую страницу, иначе на экране появлялись невменяемые «кракозябры». При этом числа от 32 до 127 всё также были зарезервированы для латиницы, независимо от кодовой страницы.

К счастью, в наши дни таких проблем практически не осталось благодаря новому стандарту Unicode, уверенно шагающему по планете. Согласно этому стандарту, для хранения символа отводится не один, а целых два байта (значения от 0 до 65535), позволяя дать каждому символу из любого мыслимого языка свой уникальный номер. С повсеместным введением Unicode исчезнет само понятие кодовых страниц и, будем надеяться, проблем, с ними связанных (по крайней мере, до тех пор, пока человек не познакомится с остальными обитателями Галактики и обилием их языков).

Однако не всё так радужно с юникодом, как нам хотелось бы. Конечно, в новых языках программирования, таких как C# или Java, символы по умолчанию являются юникодными, и большинство адептов этих языков искренне не понимают сути проблем с кодировками. К сожалению, в случае программирования микроконтроллеров на С всё обстоит не так гладко. Во-первых, язык С разрабатывался во времена, когда юникода ещё не было даже в планах. Конечно, современные компиляторы поддерживают использование юникода в программах на С, однако, это скорее костыли, нежели нормальный инструмент. Во-вторых, сами контроллеры накладывают на нас ограничения объёмами своей памяти. Часто бывает так, что памяти не хватает даже для скомпилированного кода, который приходится оптимизировать так, чтобы он влез в отведённые килобайты. На этом фоне выделение двух байт на каждый символ выглядит просто непростительно. По этим двум причинам нам придётся использовать старые добрые ANSI кодировки (особенно попотеть придётся при выводе русских символов на LCD-дисплей).

Строки

Вполне логично, что возможности по работе с текстом в С не ограничиваются отдельными символами. Было бы довольно изнурительно выводить на экран информацию по одной букве. Именно для этого и была введена концепция *строки*. Строка — это просто последовательность символов (значений типа *char*). В конце этой последовательности обязательно должен стоять *нулевой символ* — '\0'. Начало строки представляется указателем на её первый символ. Затем последующие символы считываются автоматически до тех пор, пока не встретится нулевой символ. Следующий код иллюстрирует работу со строками.

```
char s1[13] = "Hello world!";
char s2[13] = {'H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '\0'};
char s3[] = "That's weird!";

void TransmutateString(char* str) {
	int i;

	for (i = 0; i < strlen(str); i++)
		str[i] += 2;
}

void main() {
	TransmutateString(s1);
}
```

В этом коде показано несколько приёмов. Во-первых, это различные способы задания строк. Переменной s1 строка присваивается с помощью двойных кавычек. В переменной s2 мы явным образом задаём массив символов, составляющих ту же строку. Обратите внимание, что в этом случае мы вручную добавляем нулевой символ, тогда как при использовании двойных кавычек компилятор делает это автоматически. Также, и в первом и во втором случае мы явным образом указываем размер массива, причём этот размер на единицу больше, чем длина строки (именно из-за необходимости хранить нулевой символ). В третьем же случае мы не указываем размер массива, компилятор делает это за нас при сборке программы.

Мы также написали функцию *TransmutateString*, которая производит некоторые операции над каждым символом строки. Для передачи строки в виде параметра мы должны объявить этот параметр как указатель типа *char*. Если вы вспомните связь между массивами и указателями, то поймёте, что строка ничем не отличается от массива, поэтому мы можем обратиться к ней просто по имени переменной, например, *s1*.

Ну и наконец, функция *strlen*, которая используется в *TransmutateString,* является библиотечной и возвращает длину указанной строки без учёта нулевого символа.

Препроцессор

Помимо стандартных средств программирования, присущих большинству языков, в C имеется ещё один мощный инструмент — препроцессор. Он является в некотором роде метаязыком: обычные конструкции C контролируют ход выполнения программы, а препроцессор определяет то, как будет выглядеть исходный код программы ещё до компиляции. К примеру, на этапе компиляции при выполнении заданных заранее условий могут динамически включаться или отключаться целые блоки кода.

Препроцессор можно задействовать, используя *директивы*. *Директивы препроцессора* похожи на обычные ключевые слова, отличить их можно по символу октоторпа (“#”). Далее мы рассмотрим наиболее часто применяющиеся директивы.

#define

Директива *#define* позволяет создавать *макросы*. Макрос — это некий идентификатор, в соответствие которому поставлено какое-то значение. При компиляции программы все макросы в файлах заменяются на их непосредственные значения, и лишь затем начинается сам процесс компиляции. Рассмотрим пример.

```
#define THREE      3

int main()
{
      int foo = THREE; // Translates into “int foo = 3;”

      return 0;
}
```

Здесь мы создали самый простой макрос с именем *THREE*, который имеет значение *3*. Теперь препроцессор, который запускается перед компилятором, будет заменять все вхождения макроса *THREE* на цифру *3*. Здесь стоит заметить несколько важных моментов.

1. Для имени макроса действуют те же правила, что и для имён переменных. Однако для удобства макросам обычно дают имена в верхнем регистре, чтобы не путать их с обычными функциями и переменными.
2. Макрос состоит из трёх частей: директивы *#define*, имени макроса и подставляемого значения. Каждая из частей отделяется от других одним или несколькими пробелами или символами табуляции. Роль играют только первые два пробела, все остальные пробелы учитываются как значение для подстановки. Например, вместо следующего макроса будет подставляться значение “one two three”:

```
#define ONE_TWO_THREE one two three
```

3. После определения макроса точка с запятой **не ставится**. Символом окончания подстановочного значения является перенос строки.

С помощью макросов можно объявлять числовые константы.

```
#define ARRAY_SIZE       10

int main()
{
      int foo[ARRAY_SIZE];
      int i;

      for (i = 0; i < ARRAY_SIZE; ++i)
            foo[i] = i;

      return 0;
}
```

В этом примере мы используем размер массива в двух случаях: при объявлении самого массива и в цикле *for*. Если бы мы указали размер явным образом и в какой-то момент захотели изменить количество элементов массива, нам пришлось бы изменять значение сразу в двух местах. Очень часто бывают случаи, когда одно и то же значение используется в разных местах кода, и использование макросов позволяет быстро изменить это значение и не допустить ошибок.

Макросы также имеют и более сложные применения.

```
#define SB(a,b)     (a) |= 1 << (b)

int main()
{
      int foo = 2;

      SB(foo, 2); // Same as writing “foo |= 1 << 2;”

      return 0;
}
```

В этом примере мы объявили макрос, который может принимать аргументы. В этом плане макросы могут быть похожи на функции, за тем лишь исключением, что аргументы макросов не имеют типа и, по сути, являются обычным текстом для подстановки. Как видно из примера, макрос устанавливает в единицу указанный бит указанной переменной.

#ifdef, #else и #endif

Директива *#ifdef* напоминает обычное условие *if*. Эта директива позволяет проверить, определён ли указанный макрос и, если это так, включить определённый блок кода.

```
#define FOO  1

int main()
{
#ifdef FOO
      int foo = 5;
#endif

#ifdef BAR
      int bar = 5;
      DestroyTheWorld();
#else
      int bar = 10;
#endif

      int res = foo + bar;      // ret == 15

      return 0;
}
```

Здесь мы сначала определяем макрос *FOO*, а затем в функции *main* проверяем его наличие с помощью директивы *#ifdef*. Так как макрос определён, блок кода, заключённый между директивами *#ifdef* и *#endif* будет обработан компилятором.

Далее мы проверяем, объявлен ли макрос *BAR*. Так как никто его не объявлял, компилятором будет обработан блок кода, заключённый между директивами *#else* и *#endif*, в то время как блок между *#ifdef* и *#else* будет проигнорирован.

#undef

Существует также директива *#undef*, которая позволяет отменить объявление макроса.

```
#define FOO  1

int main()
{
#ifdef FOO
	int foo = 5;
#endif

	foo++;

#undef FOO

	foo--;

#ifdef FOO
	DestroyTheWorld();
#endif

	return 0;
}
```

Во время первой проверки макрос *FOO* ещё определён, поэтому объявление переменной *foo* будет обработано компилятором. Затем следует директива *#undef* и во время второй проверки макрос *FOO* будет уже не определён и соответствующий блок кода будет проигнорирован.

Структура программы на С

Во всех примерах, которые мы до сих пор рассматривали, предполагалось, что вся программа состоит только из одного файла, в котором находится весь код. Этот сценарий имеет право на жизнь, однако, почти никогда не применяется. Программа может быть очень сложной и состоять из сотен функций. Понятно, что в этом случае хранить весь код в одном файле — чистое безумие. В этом разделе мы рассмотрим способы, с помощью которых код программы можно разместить в разных файлах.

Два типа файлов: *.h и *.c

В обычную программу на С, как правило, входят два типа файлов: *заголовочные файлы* (с расширением *.h) и *файлы кода* (с расширением *.c). Обычно, эти файлы используются в паре. Например, если у вас есть файл с кодом под названием “*functions.c*“ и вы хотите использовать функции из этого файла в других местах, вам также нужно создать заголовочный файл “*functions.h*” (на самом деле, имена могут быть любыми, но логичнее давать файлам одинаковые имена).

Заголовочные файлы являются своего рода картой или «оглавлением», в котором перечислены «возможности», реализуемые соответствующим файлом кода. В разделе, посвящённом функциям, мы узнали, что для каждой функции должен быть объявлен прототип, определяющий её имя, параметры и тип возвращаемого значения. Если вы хотите использовать функцию из файла кода в других местах, то прототип этой функции помещается не сам файл кода (как в предыдущих примерах), а в соответствующий заголовочный файл.

Директива #include

Итак, мы разделили прототипы функций и их реализацию, но что дальше? Теперь мы должны включить заголовочный файл во все файлы, в которых мы собираемся использовать его функции. Именно для этого и служит директива *#include*. При обработке препроцессором вместо этой директивы подставляется содержимое файла, указанного в этой директиве. Рассмотрим пример.

Functions.h:

```
#ifndef _FUNCTIONS_H
#define _FUNCTIONS_H

int foo(int arg1, int arg2);
int bar();
void baz();

#endif
```

Functions.c:

```
#include “functions.h”

int foo(int arg1, int arg2) {
      return arg1 + arg2;
}
int bar() {
      return 42;
}
void baz() {
      sleep(1000);
}
```

main.c

```
#include "functions.h"

int main()
{
	int res = foo(2, 5);
	int answerToAllQuestions = bar();

baz();

return 0;
}
```

Итак, в этом примере есть три файла: *functions.h*, *functions.c* и *main.c*. Подразумевается, что в *functions.c* находятся некоторые служебные функции, которые мы хотим использовать в основном файле *main.c*.

В заголовочном файле *functions.h* расположены прототипы служебных функций. Благодаря этому файлу компилятор будет «знать», какие функции ему доступны. Обратите внимание на хитрую конструкцию из директив препроцессора, в которую обёрнуто содержимое заголовочного файла. Эта конструкция нужна для того, чтобы при включении заголовочных файлов в разные места компилятор обработал его содержимое только один раз. Этот нюанс довольно сложно объяснить простым языком, поэтому, нужно просто запомнить, что содержимое заголовочного файла всегда надо оборачивать в подобную конструкцию (имя макроса может быть любым, но обязательно уникальным и неповторяющимся в других заголовочных файлах).

В файле кода *functions.c* мы первым делом включаем наш заголовочный файл. Таким образом, мы, по сути, вставляем в самое начало файла наши прототипы, определённые в заголовочном файле. Далее мы просто реализуем объявленные функции, написав для них необходимый код.

Файл *main.c* является основным для нашей программы, так как в нём расположена функция *main*. Мы хотим использовать в ней наши служебные функции, поэтому включаем заголовочный файл и сюда. Это, опять же, равносильно копированию прототипов функций и вставке их в начало файла. Теперь, имея объявленные прототипы, мы можем использовать функции из *functions.c* так, как если бы они находились в самом файле *main.c*. Всё, что нужно — это поместить в *main.c* прототипы нужных функций, а о поиске непосредственного кода этих функций позаботится компилятор.

Создание сложной структуры

В заголовочных файлах можно размещать не только прототипы функций, но и объявления макросов, структур и других типов данных. Кроме этого, заголовочные файлы можно включать не только в файлы кода, но и в другие заголовочные файлы.

Macro.h

```
#ifndef _MACRO_H
#define _MACRO_H

#define UINT unsigned long int
#define UCHAR	unsigned char

#endif
```

Functions.h

```
#ifndef _FUNCTIONS_H
#define _FUNCTIONS_H

#include “Macro.h”

UINT UnsignedAdd(UINT a, UINT b);
UCHAR UnsignedAddChar(UCHAR a, UCHAR b);

#endif
```

Functions.c

```
#include “functions.h”

UINT UnsignedAdd(UINT a, UINT b)
{
	return a + b;
}

UCHAR UnsignedAddChar(UCHAR a, UCHAR b)
{
	return a + b;
}
```

main.c

```
#include “Macro.h”
#include “Functions.h”

int main()
{
	UINT resI = UnsignedAdd(17000 + 18000);
	UCHAR resC = UnsignedAddChar(127 + 129);

	return 0;
}
```

В этом примере мы создали макросы *UINT* и *UCHAR* для того, чтобы сократить длинную запись беззнаковых типов *int* и *char*. Затем мы включили заголовок *Macro.h* в другой заголовок *Functions.h*, в котором объявили функции, использующие новые «типы». Далее мы пошли по уже проторённой дорожке: реализовали служебные функции в *functions.c* и использовали их в *main.c*. Строго говоря, мы могли бы не включать *Macro.h* в *main.c*, так как этот заголовок уже содержится во включённом *functions.h*. Однако мы сделали это для наглядности, так как не всегда бывает удобно держать в голове структуру заголовков и вещи, которые в них объявлены. Компилятор же проигнорирует «лишнее» включение именно из-за той самой препроцессорной «обёртки».

Заключение

На этом мы закончим краткий обзор языка C. Безусловно, за бортом осталось много неосвещённых тем. Некоторые из них будут обсуждаться в описаниях к лабораторным работам, а некоторые, наиболее сложные и не входящие в обязательную программу, можно изучить самостоятельно с помощью книг из списка литературы. Напоследок хотелось бы сделать несколько небольших, но важных замечаний.

Вопреки расхожему мнению, недостаточно написать код, который просто работает. Всегда нужно стремиться к коду, который можно легко понять и изменить. Один из гуру программирования сказал по этому поводу: «Пишите код так, как если бы после вас его поддерживал психопат, склонный к насилию и знающий, где вы живёте». Хороший код

складывается из многих элементов. Вот несколько советов, которые могут вам пригодиться:

1. Пишите комментарии. Это не значит, что нужно комментировать каждую строку и подробно описывать тривиальные вещи. Однако комментарии должны давать общее представление о том, что делает код, и как он работает. Не стесняйтесь писать длинные комментарии. Если они действительно содержат полезную информацию, вам только скажут «спасибо».

2. Разбивайте код на функции. Если вы видите, что какой-то участок кода решает вполне определённую и логически независимую задачу, этот участок вполне можно выделить в функцию. Такой подход упрощает понимание программы и позволяет быстрее вносить в неё изменения. Если вы опасаетесь за издержки производительности, связанные с вызовом функций, вы можете объявлять их с ключевым словом *inline*. В этом случае компилятор будет подставлять тело функции в места, где она вызывается (аналогично макросам). Однако в большинстве случаев такой подход излишен, так как компилятор самостоятельно проводит необходимые оптимизации.

3. Используйте локальные переменные, когда это возможно. Глобальные переменные ухудшают восприятие кода и повышают вероятность появления трудноуловимых багов. Кроме этого, глобальные переменные постоянно занимают отведённую им память, что в условиях ограниченности ресурсов микроконтроллера является серьёзным недостатком. В большинстве случаев глобальные переменные легко заменяются передачей переменных в качестве аргументов функций.

4. Применяйте константы для часто используемых значений. Мы уже обсуждали этот вопрос в разделе, посвящённом макросам. К этому можно также добавить тот факт, что именованные константы дают дополнительные подсказки о том, что делает конкретный код. Однако не стоит увлекаться этим правилом: однажды в одном из проектов я встретил константу “THOUSAND = 1000”. Для меня до сих пор остаётся загадкой, что имел в виду автор. Возможно, он хотел предусмотреть в своей программе абсолютно всё, вплоть до изменения фундаментальных законов мироздания.

5. Форматируйте код. Применяйте отступы для блоков кода, заключённых в фигурные скобки. Код с отступами читается гораздо легче, чем «спагетти» из операторов, функций и управляющих конструкций (а если там, к тому же, нет ни одного комментария, то вообще замечательно). Всегда используйте один и тот же стиль форматирования. Например, существуют разные стили оформления блоков с кодом: кто-то ставит открывающую фигурную скобку на одной строке с ключевым словом или функцией, а кто-то — на следующей. Это никоим образом не влияет на результирующий код, и каждый волен использовать стиль, который ему больше по душе. Самое главное — придерживаться этого стиля во всех случаях. В этом пособии мы смешиваем разные стили, однако, наша карма остаётся чистой, так как мы делаем это только для того, чтобы оптимально разместить блоки кода на печатных страницах. Не берите с нас пример, вам экономить нечего.