15

Манипулирование битами

В ЭТОЙ ГЛАВЕ...

- Операции: `~`, `&`, `|`, `^`, `>>`, `<<`, `&=`, `|=`, `^=`, `>>=`, `<<=`
- Обзор двоичной, восьмеричной и шестнадцатеричной систем счисления
- Два средства языка C для обработки отдельных битов значения: побитовые операции и битовые поля
- Ключевые слова: `_Alignas`, `_Alignof`

Язык C позволяет управлять индивидуальными битами значения переменной. Может возникнуть вопрос: для чего это нужно? Не сомневайтесь, что иногда такая возможность необходима или, по крайней мере, удобна. Примером может служить управление некоторым физическим устройством, что часто связано с передачей нескольких битов, причем каждый из них имеет определенный смысл. Кроме того, информация о файлах в операционной системе обычно хранится в виде определенных битов, указывающих на отдельные элементы. Многие операции сжатия и шифрования связаны с управлением битами. Языки высокого уровня, как правило, не обеспечивают такого уровня детализации. Способность совмещать возможности языка высокого уровня с операциями на уровне, который обычно оставляется за языком ассемблера, делает C предпочтительным выбором для написания драйверов устройств и встраиваемого кода.

В этой главе мы исследуем возможности языка C по работе с битами, первоначально ознакомившись с понятиями бита, байта, двоичной и других систем счисления.

Двоичные числа, биты и байты

Обычная форма записи чисел основана на числе 10. Например, число 2157 в позиции тысяч содержит цифру 2, в позиции сотен — 1, в позиции десятков — 5, а в позиции единиц — 7. Это означает, что число 2157 можно рассматривать следующим образом:

$$2 \times 1000 + 1 \times 100 + 5 \times 10 + 7 \times 1!$$

Принимая во внимание, что 1000 — это 10 в кубе, 100 — это десять в квадрате, 10 — 10 в первой степени, а 1 — это 10 (как и любое другое положительное число) в нулевой степени, число 2157 можно записать так:

$$2 \times 10^3 + 1 \times 10^2 + 5 \times 10^1 + 7 \times 10^{0!}$$

Поскольку привычная система записи чисел основана на степенях 10, мы говорим, что число 2157 записано *по основанию 10*.

Люди пользуются десятичной системой счисления потому, что у них на руках 10 пальцев. Тогда будем считать, что у бита только два пальца, т.к. он может быть установлен лишь в 0 или 1 (выключен или включен). Таким образом, для компьютера естественной является двоичная система счисления. В ней для записи чисел используются степени 2, а не 10. Числа, выраженные по основанию 2, называют *двоичными*. Число 2 играет такую же роль в двоичной системе, как число 10 в десятичной. Например, двоичная запись 1101 означает:

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

В десятичной записи это становится следующим:

$$1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 = 13$$

В двоичной системе можно представить любое целое число (при достаточном количестве битов) в форме комбинации нулей и единиц. Эта система очень удобна для цифровых вычислительных систем, у которых информация выражается в виде комбинаций включенных и выключенных состояний, что можно интерпретировать как единицы и нули. Давайте посмотрим, как двоичная система работает с однобайтовым целым числом.

Двоичные целые числа

Обычно байт содержит 8 битов. Вспомните, что в языке C термин *байт* применяется для обозначения размера памяти, используемой для хранения набора символов системы, поэтому в C байт может содержать 8, 9, 16 и другое количество битов. Однако в характеристиках модулей памяти и систем передачи данных предполагается, что байт содержит 8 битов. Чтобы излишне не усложнять, в этой главе предполагается 8-битовый байт. (Для ясности в мире вычислений 8-битовый байт часто обозначается термином *октет*.) Можно считать, что биты в байте пронумерованы справа налево с 0 до 7. Седьмой бит называется *старшим*, а нулевой бит — *младшим*. Каждый номер бита соответствует определенной степени числа 2. Такое представление байта иллюстрируется на рис. 15.1.

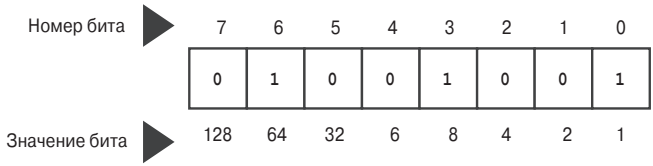

Рис. 15.1. *Номера и значения битов*

Здесь значение 128 представляет собой 2 в степени 7 и т.д. Байт имеет наибольшее значение, когда все его биты установлены в 1: 11111111. Значение этого двоичного числа определяется следующим образом:

```
128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255
```

Наименьшему значению соответствует комбинация 00000000, или просто 0. Байт может хранить числа от 0 до 255, что составляет 256 возможных значений. Или, интерпретируя комбинацию битов по-другому, программа может применять байт для хранения чисел от –128 до 127, что также дает 256 возможных значений. Например, тип `unsigned char` обычно применяет байт для представления диапазона чисел от 0 до 255, а тип `signed char` — для диапазона от –128 до 127.

Целые числа со знаком

Представление целых чисел со знаком определяется оборудованием, а не языком C. Пожалуй, самый простой способ представления чисел со знаком заключается в резервировании бита, такого как старший, для обозначения знака. В однобайтовом значении для представления самого числа остается 7 битов. В таком представлении *величины со знаком* комбинация 10000001 будет соответствовать числу –1, а комбинация 00000001 — числу 1. Тогда диапазон представляемых значений будет простираться от –127 до +127.

Один из недостатков такого подхода состоит в возможности двоякого представления нуля: +0 и -0. Это вызывает путаницу и приводит к использованию двух комбинаций битов для представления одного значения.

Метод *дополнения до двух* устраняет эту проблему, и в настоящее время он распространен наиболее широко. Мы обсудим его применительно к однобайтовому значению. В данном контексте значения от 0 до 127 представляются последними семью битами со старшим битом, установленным в 0. Пока что нет отличий от представления

величины со знаком. Точно так же, если старший бит равен 1, то число является отрицательным. Отличие начинается при определении значения этого отрицательного числа. Для этого понадобится вычесть комбинацию битов отрицательного числа из 9-битовой комбинации 100000000 (двоичного представления числа 256), в результате получив модуль значения. Для примера предположим, что комбинация имеет вид 10000000. Как байт без знака, это соответствует числу 128. Как значение со знаком, оно является отрицательным (бит 7 равен 1) и имеет величину 100000000–10000000, или 10000000 (т.е. 128). Следовательно, число равно –128. (В представлении величины со знаком оно было бы равно –0.) Подобным же образом, комбинация 10000001 соответствует значению –127, а комбинация 11111111 — значению –1. Данный метод позволяет представлять числа в диапазоне от –128 до 127.

Простейший способ смены знака двоичного числа, которое представлено методом дополнения до 2, предусматривает инвертирование каждого бита (превращение 0 в 1 и 1 в 0) и затем добавление 1. Поскольку 1 — это 00000001, то –1 соответствует 11111110 + 1, или 11111111, как уже было показано.

Метод *дополнения до единицы* формирует отрицательное число путем инвертирования каждого бита в комбинации. Например, комбинация 000000001 — это 1, а 11111110 — значение –1. Этот метод также имеет –0: 11111111. Диапазон представляемых чисел (для однобайтового значения) составляет от –127 до +127.

Двоичные числа с плавающей запятой

Числа с плавающей запятой хранятся в виде двух частей: двоичной дроби и двоичной экспоненты. Давайте посмотрим, как это происходит.

Двоичные дроби

Десятичная дробь 0.527 является следующей суммой:

```
5/10 + 2/100 + 7/1000
```

Здесь знаменатели представляют возрастающие степени 10. В двоичной дроби знаменатели будут степенями 2. Таким образом, двоичная дробь .101 может быть записана так:

```
1/2 + 0/4 + 1/8
```

В десятичной записи это имеет вид:

```
0.50 + 0.00 + 0.125
```

или 0.625.

Многие дроби, такие как 1/3, не могут быть точно представлены в десятичной записи. Аналогично, многие дроби невозможно точно представить и в двоичной записи. На самом деле точно могут быть представлены лишь комбинации составляющих, которые кратны степеням 1/2. Таким образом, дроби 3/4 и 7/8 можно точно записать в двоичном представлении, но дроби 1/3 и 2/5 — нельзя.

Представление чисел с плавающей запятой

Представление числа с плавающей запятой в компьютере предусматривает выделение некоторого количества (в зависимости от системы) битов для хранения двоичной дроби. Дополнительные биты представляют экспоненту. В общих терминах действительное значение числа определяется как произведение двоичной дроби на 2 в степени, выраженной экспонентой. Умножение числа с плавающей запятой, скажем, на 4, увеличивает экспоненту в 2 раза, оставляя двоичную дробь неизменной. Умножение на число, не являющееся степенью 2, изменяет двоичную дробь и при необходимости экспоненту.

Другие основания систем счисления

Специалисты в области компьютеров часто используют системы счисления с основаниями 8 и 16. Поскольку числа 8 и 16 являются степенями 2, эти системы счисления более тесно связаны с двоичной системой компьютера, чем десятичная система.

Восьмеричная система счисления

Восьмеричной называется система счисления с основанием 8. В этой системе каждое знакоместо в числе представляет степень 8. Для записи применяются цифры от 0 до 7. Например, восьмеричное число 451 (в C записывается как 0451) представлено следующим образом:

$$4 \times 8^2 + 5 \times 8^1 + 1 \times 8^0 = 297 \text{ (по основанию 10)}$$

Каждая восьмеричная цифра соответствует трем двоичным цифрам (табл. 15.1). Такое соответствие упрощает перевод чисел между системами. Например, восьмеричное число 0377 — это двоичное число 11111111. Отбросив ведущий 0, мы заменяем 3 комбинацией 011, после чего каждую цифру 7 заменяем 111. Единственное неудобство состоит в том, что трехзначное восьмеричное число в двоичной форме может занимать до 9 битов. Поэтому восьмеричное значение, превышающее 0377, требует более одного байта. Обратите внимание, что внутренние нули не опускаются: числу 0173 соответствует комбинация 01 111 011, а не 01 111 11.

Таблица 15.1. Двоичные эквиваленты восьмеричных цифр

Восьмеричная цифра	Двоичный эквивалент
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Шестнадцатеричная система счисления

Шестнадцатеричной называется система систему счисления с основанием 16. В ней используются степени 16 и цифры от 0 до 15, но из-за того, что в десятичной системе отсутствуют цифры для представления значений от 10 до 15, в шестнадцатеричной системе для них применяются буквы от A до F. Например, шестнадцатеричное число A3F (в C записывается как 0xA3F) представляет следующее значение:

$$10 \times 16^2 + 3 \times 16^1 + 15 \times 16^0 = 2623 \text{ (по основанию 10)}$$

Здесь цифра A представляет значение 10, а F — значение 15. Для обозначения шестнадцатеричных цифр в C разрешено использовать буквы нижнего или верхнего регистра. Таким образом, число 2623 можно записать также в виде 0xa3f.

Каждая шестнадцатеричная цифра соответствует двоичному числу с 4 цифрами, так что две шестнадцатеричных цифры дают в точности один 8-битовый байт. Первая цифра представляет старшие 4 бита, а вторая цифра — младшие 4 бита. Это делает шестнадцатеричное представление естественным выбором для записи значений байтов. Соответствие между шестнадцатеричными цифрами и двоичными числами показано в табл. 15.2. Например, шестнадцатеричное число 0xC2 преобразуется в комбинацию 11000010. Для обратного преобразования комбинацию 11010101 необходимо представить в виде 1101 0101 и затем записать как 0xD5.

Таблица 15.2. Десятичные, шестнадцатеричные числа и их двоичные эквиваленты

Десятичное число	Шестнадцатеричная цифра	Двоичный эквивалент
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Теперь, когда вы ознакомились с понятием битов и байтов, давайте посмотрим, что в языке C можно с ними делать. Существуют два средства, помогающие манипулировать битами. Первое — это набор из шести побитовых операций, которые воздействуют на биты. Второе средство — это форма *полей* данных, которая предоставляет доступ к битам внутри значения `int`. Эти средства обсуждаются в последующих разделах.

Побитовые операции

Язык C предлагает два вида побитовых операций: логические операции и операции сдвига. В последующих примерах мы будем записывать значения в двоичной системе, чтобы вы могли видеть, что происходит с битами. В действительной программе вы будете применять целочисленные переменные или константы в обычных формах. Например, вместо `00011001` будет использоваться запись `25`, `031` или `0x19`. В рассматриваемых примерах мы будем применять 8-битовые числа с нумерацией битов слева направо от 0 до 7.

Побитовые логические операции

Четыре логических побитовых операции работают с целочисленными данными, включая тип `char`. Они называются *побитовыми* потому, что выполняются над каждым битом независимо от бита, находящегося слева или справа. Не путайте их с обычными логическими операциями (`&&`, `||` и `!`), которые имеют дело со значениями целиком.

Дополнение до единицы или побитовое отрицание: ~

Унарная операция ~ преобразует каждую единицу в ноль, а каждый ноль в единицу, как показано в следующем примере:

```
~(10011010)   // выражение
(01100101)    // результат
```

Предположим, что переменной `val` типа `unsigned char` присвоено значение `2`. В двоичном виде `2` имеет вид `00000010`. Тогда `~val` будет иметь значение `11111101`, или `253`. Обратите внимание, что операция не изменяет значения переменной `val`, в точности как не изменяет значение `val` выражение `3 * val`; значением `val` по-прежнему является `2`, но создается новое значение, которое можно использовать или присваивать где-то в другом месте:

```
newval = ~val;
printf("%d", ~val);
```

Если вы хотите изменить значение `val` на `~val`, применяйте следующий простой оператор присваивания:

```
val = ~val;
```

Побитовая операция "И": &

Двоичная операция `&` создает новое значение за счет выполнения побитового сравнения двух операндов. Для каждой позиции результирующий бит будет равен 1, только если оба соответствующих бита в операндах равны 1. (В терминах истинный/ложный можно сказать, что результат будет истинным, только когда каждый из двух битовых операндов является истинным.) Таким образом, в результате вычисления выражения

```
(10010011) & (00111101)    // выражение
```

получается следующее значение:

```
(00010001)                 // результат
```

Причина в том, что только нулевой и четвертый биты равны `1` в обоих операндах. В C также имеется операция "И", объединенная с присваиванием: `&=`.

Оператор

```
val &= 0377;
```

дает такой же результат, как и следующий оператор:

```
val = val & 0377;
```

Побитовая операция "ИЛИ": |

Двоичная операция `|` создает новое значение за счет выполнения побитового сравнения двух операндов. Для каждой позиции бит будет равен 1, если любой из соответствующих битов в операндах равен 1. (В терминах истинный/ложный можно сказать, что результат будет истинным в случае, когда один или другой битовый операнд является истинным либо сразу оба.)

Таким образом, в результате вычисления выражения

```
(10010011) | (00111101)    // выражение
```

получается следующее значение:

```
(10111111)                 // результат
```

Причина в том, что биты во всех позициях кроме 6 имеют значение 1 в одном или в другом операнде (или в обоих). В C также существует операция "ИЛИ", объединенная с присваиванием: |=. Оператор

```
val |= 0377;
```

дает тот же результат, что и следующий оператор:

```
val = val | 0377;
```

Побитовое "исключающее ИЛИ": ^

Двоичная операция ^ выполняет побитовое сравнение двух операндов. Для каждой позиции результирующий бит будет равен 1, если один или другой (но не оба) из соответствующих битов в операндах равен 1. (В терминах истинный/ложный можно сказать, что результат будет истинным в случае, когда один или другой битовый операнд является истинным, но не оба.) Таким образом, в результате вычисления выражения

```
(10010011) ^ (00111101)    // выражение
```

получается следующее значение:

```
(10101110)                 // результат
```

Обратите внимание, что поскольку бит 0 равен 1 у обоих операндов, результирующий бит 0 получает значение 0.

В языке C также имеется операция "исключающее ИЛИ", объединенная с присваиванием: ^=. Оператор

```
val ^= 0377;
```

дает тот же результат, что и следующий оператор:

```
val = val ^ 0377;
```

Случай применения: маски

Побитовая операция "И" часто используется с маской. *Маска* — это комбинация битов, в которой некоторые биты включены (1), а некоторые выключены (0). Чтобы понять, почему ее так назвали, давайте посмотрим, что происходит, когда мы объединяем какую-то величину с маской с применением операции &. Для примера предположим, что вы определили символическую константу MASK как 2 (т.е. 00000010), у которой ненулевым является только бит с номером 1. Тогда оператор

```
flags = flags & MASK;
```

приведет к установке всех битов flags (кроме первого) в 0, т.к. любой бит, объединяемый с 0 посредством операции "И", дает 0. Бит номер 1 переменной остается неизменным. (Если бит равен 1, то значением 1 & 1 будет 1; если же бит равен 0, то 0 & 1 дает 0.) Такой процесс называется "использованием маски", поскольку нули в маске скрывают соответствующие биты в переменной flags.

Развивая аналогию, биты с 0 в маске можно считать непрозрачными, а биты с 1 — прозрачными. Выражение flags & MASK похоже на накрывание маской комбинации битов flags; видимыми из-под маски будут только те биты, которым в MASK соответствуют биты с 1 (рис. 15.2).

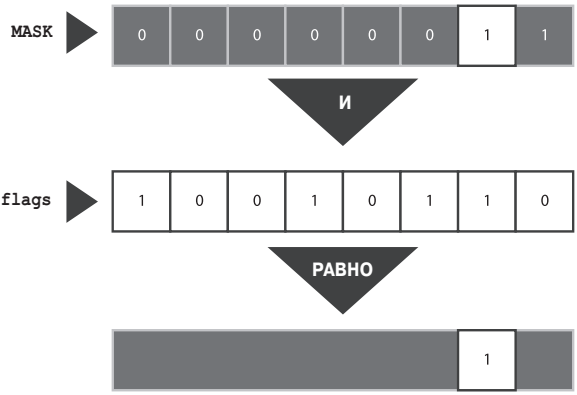

Рис. 15.2. *Наглядное представление маски*

Для сокращения кода можно применить операцию “И”, объединенную с присваиванием:

```
flags &= MASK;
```

Ниже показан один распространенный случай использования этой операции:

```
ch &= 0xff;  /* или ch &= 0377; */
```

Вспомните, что значение `0xff` записывается как `11111111` в двоичном или как `0377` в восьмеричном виде. Эта маска оставляет последние восемь битов в `ch` без изменений, а остальные устанавливает в 0. Независимо от того, сколько битов содержит исходная переменная `ch` — 8, 16 или более, — финальное значение усекается до величины, которая умещается в один 8-битовый байт. В данном случае маска имеет ширину 8 битов.

Случай применения: включение (установка) битов

Иногда требуется включить отдельные биты в значении, оставив остальные без изменений. Например, компьютер IBM PC управляет оборудованием, отправляя нужные значения в порты. Для активизации, скажем, динамика, необходимо включить бит 1, а остальные биты оставить неизменными. Этого можно сделать с помощью побитовой операции “ИЛИ”.

Например, пусть имеется константа `MASK`, в которой бит 1 установлен в `1`. Тогда оператор

```
flags = flags | MASK;
```

включает бит номер 1 в переменной `flags` и оставляет все остальные биты без изменений. Это объясняется тем, что любой бит, объединенный с 0 посредством операции `|`, остается самим собой, а объединенный с 1 с использованием `|`, становится равным 1.

Например, пусть `flags` равно `00001111` и `MASK` — `10110110`. Выражение

```
flags | MASK
```

становится

```
(00001111) | (10110110)    // выражение
```

и после вычисления дает следующий результат:

```
(10111111)                 // результат
```

Все биты, установленные в 1 внутри MASK, также будут установлены в 1 в результате. Все биты в flags, которые соответствуют битам 0 в MASK, остаются неизмененными.

Для краткости можно использовать побитовую операцию "ИЛИ", объединенную с присваиванием:

```
flags |= MASK;
```

Этот оператор установит в 1 те биты falgs, которым соответствуют включенные биты в MASK, оставив другие биты без изменений.

Случай применения: выключение (очистка) битов

Точно так же, как удобно иметь возможность включать отдельные биты, не затрагивая остальные, не менее удобно располагать возможностью их выключения. Предположим, что требуется отключить бит номер 1 в переменной flags. И снова MASK имеет включенный только бит 1. Можно воспользоваться следующим оператором:

```
flags = flags & ~MASK;
```

Поскольку в MASK все биты кроме бита 1 выключены, выражение ~MASK дает значение, в котором все биты кроме бита 1 включены. Объединение 1 с любым битом, используя операцию &, дает сам этот бит, поэтому оператор оставляет все биты кроме бита 1 без изменений. Объединение 0 с любым битом посредством операции & дает 0 независимо от исходного значения бита.

Например, пусть flags равно 00001111 и MASK — 10110110. Выражение

```
flags & ~MASK
```

становится

```
(00001111) &^ (10110110)      // выражение
```

и после вычисления дает следующий результат:

```
(00001001)                    // результат
```

Все биты, установленные в 1 внутри MASK, будут установлены в 0 в результате. Все биты в flags, которые соответствуют битам 0 в MASK, остаются неизмененными.

Ниже представлена сокращенная форма:

```
flags &= ~MASK;
```

Случай применения: переключение битов

Переключение бита означает его выключение, если он включен, и включение, если выключен. Для переключения битов можно применять побитовую операцию исключающего "ИЛИ". Идея в том, что если b — это установленное состояние бита (1 или 0), то 1 ^ b равно 0, когда b равно 1, и 1, когда b равно 0. Кроме того, выражение 0 ^ b дает b независимо от значения b. Следовательно, в результате объединения значения с маской с использованием операции ^ биты, соответствующие 1 в маске, переключаются, а биты, соответствующие 0 в маске, останутся неизмененными. Чтобы переключить бит 1 переменной flags, можно выполнить одно из следующих действий:

```
flags = flags ^ MASK;
flags ^= MASK;
```

Например, пусть flags равно 00001111 и MASK — 10110110. Выражение

```
flags ^ MASK
```

становится

```
(00001111) ^ (10110110)  // выражение
```

и после вычисления дает следующий результат:

```
(10111001)                  // результат
```

Все биты, установленные в 1 внутри MASK, приводят к переключению соответствующих битов в flags. Все биты в flags, которые соответствуют битам 0 в MASK, остаются неизмененными.

Случай применения: проверка значения бита

Вы уже видели, как изменять значения битов. Предположим, что вместо этого нужно проверить значение какого-нибудь бита. Например, установлен ли в 1 бит 1 в flags? Простое сравнение flags и MASK здесь не подойдет:

```
if (flags == MASK)
   puts("Совпадает!");   /* не работает */
```

Даже если бит 1 переменной flags установлен в 1, значение какого-то другого бита в flags может сделать результат сравнения недействительным. Чтобы выполнить сравнение только бита 1 в flags с MASK, необходимо сначала замаскировать остальные биты flags:

```
if ((flags & MASK) == MASK)
   puts("Совпадает!");
```

Побитовые операции имеют приоритет ниже, чем у операции ==, поэтому выражение flags & MASK должно быть заключено в скобки.

Во избежание неполного охвата информации, битовая маска должна иметь ширину не меньше, чем у маскируемого значения.

Побитовые операции сдвига

Теперь давайте взглянем на операции сдвига в языке C. Побитовые операции сдвига сдвигают биты влево или вправо. Для большей наглядности мы здесь также будем применять двоичную запись чисел.

Сдвиг влево: <<

Операция сдвига влево (<<) сдвигает биты значения левого операнда влево на количество позиций, заданное правым операндом. Освобождаемые позиции заполняются 0, а биты, выходящие за пределы значения левого операнда, теряются. В следующем примере каждый бит сдвигается на две позиции влево:

```
(10001010) << 2          // выражение
(00101000)               // результат
```

Эта операция выдает новое битовое значение, но не изменяет операнды. Для примера предположим, что переменная stonk имеет значение 1. Выражение stonk<<2 дает 4, но значением stonk по-прежнему является 1. Чтобы изменить значение переменной, можно воспользоваться операцией сдвига влево с присваиванием (<<=). Эта операция сдвигает биты переменной влево на количество позиций, указанное в правом операнде. Вот пример:

```
int stonk = 1;
int onkoo;
onkoo = stonk << 2;       /* присваивает 4 переменной onkoo */
stonk <<= 2;              /* изменяет значение stonk на 4   */
```

Сдвиг вправо: >>

Операция сдвига вправо (>>) сдвигает биты значения левого операнда вправо на количество позиций, указанное в правом операнде. Биты, которые выходят за правую границу левого операнда, теряются. Для типов без знака освобождаемые слева позиции заполняются 0. Для типов со знаком данных результат зависит от системы. Освобождаемые позиции могут заполняться 0 либо битом знака (самого левого):

```
(10001010) >> 2      // выражение, значение со знаком
(00100010)           // результат в одних системах
(10001010) >> 2      // выражение, значение со знаком
(11100010)           // результат в других системах
```

Для значения без знака результат будет следующим:

```
(10001010) >> 2      // выражение, значение без знака
(00100010)           // результат во всех системах
```

Каждый бит перемещается на две позиции вправо, а освобождаемые позиции заполняются 0.

Операция сдвига вправо с присваиванием (>>=) сдвигает вправо биты левого операнда на заданное в правом операнде количество позиций, например:

```
int sweet = 16;
int ooosw;
ooosw = sweet >> 3;  /* ooosw равно 2, sweet по-прежнему 16 */
sweet >>=3;          /* значение sweet изменилось на 2      */
```

Случай применения: побитовые операции сдвига

Побитовые операции сдвига могут служить удобным и эффективным (в зависимости от оборудования) средством выполнения умножения и деления на степени 2:

`number << n` Умножает `number` на 2 в степени `n`

`number >> n` Делит `number` на 2 в степени `n`, если значение `number` неотрицательно

Эти операции сдвига аналогичны смещению десятичной точки при умножении или делении на 10.

Операции сдвига могут также использоваться для извлечения групп битов из более крупных конструкций. Предположим, что для представления значений цвета применяется переменная типа `unsigned long`, причем младший байт содержит интенсивность красной составляющей, следующий байт — интенсивность зеленой составляющей, а третий байт — интенсивность синей составляющей цвета. Пусть необходимо сохранить интенсивность каждой составляющей в собственной переменной типа `unsigned char`. Для этого можно написать такой код:

```
#define BYTE_MASK 0xff
unsigned long color = 0x002a162f;
unsigned char blue, green, red;
red = color & BYTE_MASK;
green = (color >> 8) & BYTE_MASK;
blue = (color >> 16) & BYTE_MASK;
```

В коде посредством операции сдвига вправо 8-битовое значение составляющей цвета перемещается в младший байт. Затем с помощью приема с маской значение младшего байта присваивается желаемой переменной.

Пример программы

В главе 9 при написании программы преобразования чисел в двоичное представление мы использовали рекурсию. Теперь мы решим ту же задачу с применением побитовых операций. Программа в листинге 15.1 читает вводимое с клавиатуры целое число и передает его вместе с адресом строки в функцию по имени `itobs()`, которая строит для целочисленного значения строку с двоичным представлением. Для определения подходящей комбинации 0 и 1, помещаемой в строку, эта функция использует побитовые операции.

Листинг 15.1. Программа `binbit.c`

```
/* binbit.c -- использование операций с битами для отображения двоичного
представления чисел */
#include <stdio.h>
#include <limits.h>  // для CHAR_BIT количество битов на символ
char * itobs(int, char *);
void show_bstr(const char *);

int main(void)
{
    char bin_str[CHAR_BIT * sizeof(int) + 1];
    int number;

    puts("Вводите целые числа и просматривайте их двоичные представления.");
    puts("Нечисловой ввод завершает программу.");
    while (scanf("%d", &number) == 1)
    {
        itobs(number,bin_str);
        printf("%d представляется как ", number);
        show_bstr(bin_str);
        putchar('\n');
    }
    puts("Программа завершена.");

    return 0;
}

char * itobs(int n, char * ps)
{
    int i;
    const static int size = CHAR_BIT * sizeof(int);

    for (i = size - 1; i >= 0; i--, n >>= 1)
        ps[i] = (01 & n) + '0';  // предполагается кодировка ASCII или похожая
    ps[size] = '\0';

    return ps;
}

/* отображение двоичной строки блоками по 4 */
void show_bstr(const char * str)
{
    int i = 0;

    while (str[i])  /* пока не будет получен нулевой символ */
    {
        putchar(str[i]);
        if(++i % 4 == 0 && str[i])
            putchar(' ');
    }
}
```

В листинге 15.1 применяется макрос CHAR_BIT из заголовочного файла limits.h. Этот макрос представляет количество битов в типе char. Операция sizeof возвращает размер в терминах char, поэтому выражение CHAR_BIT * sizeof(int) дает количество битов в значении int. Массив bin_str содержит на один элемент больше этой величины, чтобы можно было добавить в него завершающий нулевой символ.

Функция itobs() возвращает тот же самый адрес, который ей был передан, так что ее вызов можно использовать, к примеру, в качестве аргумента printf(). На первой итерации цикла for функция вычисляет выражение 01 & n. Операнд 01 — это восьмеричное представление маски, у которой все биты кроме нулевого установлены в 0. Следовательно, результатом 01 & n будет значение последнего бита в n. Значением является 0 или 1, но для массива необходим *символ* '0' или *символ '1'*. Преобразование осуществляется добавлением кода для '0'. (Это предполагает, что цифры кодируются последовательно, как в ASCII.) Результат помещается в предпоследний элемент массива. (Последний элемент зарезервирован для нулевого символа.)

Кстати, вместо выражения 01 & n можно применить и 1 & n. Использование восьмеричного значения 1 вместо десятичного выглядит более стильно. С этой точки зрения вариант 0x1 & n, пожалуй, даже лучше.

Затем в цикле выполняются операторы i-- и n >>= 1. Первый оператор приводит к переходу на предыдущий элемент массива, а второй сдвигает биты в n на одну позицию вправо. На следующей итерации цикла код найдет значение нового самого правого бита. После этого соответствующий ему символ цифры помещается в элемент, предшествующий последней цифре. В подобной манере функция заполняет массив справа налево.

Для отображения результирующей строки можно применять printf() или puts(). Тем не менее, в листинге 15.1 определена функция show_bstr(), которая разбивает последовательность битов на группы по четыре, чтобы облегчить восприятие строки.

Ниже приведен пример выполнения программы:

```
Вводите целые числа и просматривайте их двоичные представления.
Нечисловой ввод завершает программу.
7
7 представляется как 0000 0000 0000 0000 0000 0000 0000 0111
2013
2013 представляется как 0000 0000 0000 0000 0000 0111 1101 1101
-1
-1 is 1111 1111 1111 1111 1111 1111 1111 1111
32123
32123 представляется как 0000 0000 0000 0000 0111 1101 0111 1011
q
Программа завершена.
```

Еще один пример

Давайте рассмотрим еще один пример. На этот раз цель заключается в том, чтобы написать функцию, которая инвертирует последние n битов в значении, принимая в качестве аргументов n и само значение.

Операция ~ инвертирует биты, но делает это со всеми битами в байте, а не только с избранными. Однако, как вы уже видели, для переключения отдельных битов можно использовать операцию ^ (исключающее "ИЛИ"). Предположим, что создана маска, в которой последние n битов установлены в 1, а остальные — в 0. Тогда применение ^ к этой маске и значению переключает, или *инвертирует*, последние n битов, оставляя остальные биты без изменений. Такой подход реализован в следующем фрагменте кода:

```
int invert_end(int num, int bits)
{
   int mask = 0;
   int bitval = 1;

   while (bits-- > 0)
   {
      mask |= bitval;
      bitval <<= 1;
   }
   return num ^ mask;
}
```

Маска создается в цикле `while`. Изначально в `mask` все биты установлены в 0. На первой итерации цикла бит 0 устанавливается в 1, после чего значение `bitval` увеличивается до 2, т.е. в нем бит 0 устанавливается в 0, а бит 1 — в 1. На следующей итерации бит 1 в `mask` устанавливается в 1 и т.д. В конце концов, операция `num ^ mask` дает желаемый результат.

Для тестирования функции ее можно внедрить в предыдущую программу, как показано в листинге 15.2.

Листинг 15.2. Программа `invert4.c`

```
/* invert4.c -- использование операций с битами для отображения двоичного
представления чисел */
#include <stdio.h>
#include <limits.h>
char * itobs(int, char *);
void show_bstr(const char *);
int invert_end(int num, int bits);
int main(void)
{
  char bin_str[CHAR_BIT * sizeof(int) + 1];
   int number;
   puts("Вводите целые числа и просматривайте их двоичные представления.");
   puts("Нечисловой ввод завершает программу.");
   while (scanf("%d", &number) == 1)
   {
      itobs(number,bin_str);
      printf("%d представляется как\n", number);
      show_bstr(bin_str);
      putchar('\n');
      number = invert_end(number, 4);
      printf("Инвертирование последних 4 битов дает\n");
      show_bstr(itobs(number,bin_str));
      putchar('\n');
   }
   puts("Программа завершена.");
   return 0;
}
char * itobs(int n, char * ps)
{
   int i;
   const static int size = CHAR_BIT * sizeof(int);
   for (i = size - 1; i >= 0; i--, n >>= 1)
      ps[i] = (01 & n) + '0';
   ps[size] = '\0';
   return ps;
}
```

```
/* отображение двоичной строки блоками по 4 */
void show_bstr(const char * str)
{
    int i = 0;

    while (str[i])  /* пока не будет получен нулевой символ */
    {
        putchar(str[i]);
        if(++i % 4 == 0 && str[i])
            putchar(' ');
    }
}

int invert_end(int num, int bits)
{
    int mask = 0;
    int bitval = 1;

    while (bits-- > 0)
    {
        mask |= bitval;
        bitval <<= 1;
    }

    return num ^ mask;
}
```

Ниже представлен пример выполнения программы:

```
Вводите целые числа и просматривайте их двоичные представления.
Нечисловой ввод завершает программу.
7
7 представляется как
0000 0000 0000 0000 0000 0000 0000 0111
Инвертирование последних 4 битов дает
0000 0000 0000 0000 0000 0000 0000 1000
12541
12541 представляется как
0000 0000 0000 0000 0011 0000 1111 1101
Инвертирование последних 4 битов дает
0000 0000 0000 0000 0011 0000 1111 0010
q
Bye!
```

Битовые поля

Второй метод манипулирования битами предусматривает использование *битового поля*, которое представляет собой просто набор соседствующих битов внутри значения типа `signed int` или `unsigned int`. (Стандарты C99 и C11 дополнительно разрешают иметь битовые поля типа `_Bool`.) Битовое поле создается путем объявления структуры, в которой помечено каждое поле и определен его размер. Например, следующее объявление устанавливает четыре однобитовых поля:

```
struct {
    unsigned int autfd   : 1;
    unsigned int bldfc   : 1;
    unsigned int undln   : 1;
    unsigned int itals   : 1;
} prnt;
```

Такое определение приводит к получению структуры prnt, содержащей четыре однобитовых поля. Теперь для присваивания значений отдельным полям можно применять обычную операцию членства в структуре:

```
prnt.itals = 0;
prnt.undln = 1;
```

Поскольку каждое из этих полей — это просто один бит, присваивать можно только значения 1 и 0. Переменная prnt хранится в ячейке памяти размером типа int, но в этом примере используются только четыре бита.

Структуры с битовыми полями служат удобным средством для отслеживания настроек. Многие настройки, такие как полужирное или курсивное начертание шрифта, сводятся к указанию одной из двух опций: "включено" или "отключено", "да" или "нет", "истинно" или "ложно". Когда нужен одиночный бит, не имеет смысла применять целую переменную. Структура с битовыми полями позволяет хранить множество настроек в одной конструкции.

Временами настройка предусматривает более двух опций, поэтому для представления всех вариантов одного бита оказывается недостаточно. Это не проблема, т.к. размеры полей не ограничены одним битом. Структуру можно определить следующим образом:

```
struct {
   unsigned int code1 : 2;
   unsigned int code2 : 2;
   unsigned int code3 : 8;
} prcode;
```

Этот код создает два 2-битовых поля и одно 8-битовое. Теперь возможны следующие присваивания:

```
prcode.code1 = 0;
prcode.code2 = 3;
prcode.code3 = 102;
```

Нужно просто следить, чтобы значение не превышало размерность поля.

А что, если общее количество объявленных битов превысит размер типа unsigned int? Тогда будет использоваться следующая область для хранения unsigned int. Отдельное поле не должно перекрывать границу между двумя смежными областями unsigned int. Компилятор автоматически сдвигает такое перекрывающее определение поля, чтобы выровнять его по границе unsigned int. Когда это происходит, в первой области unsigned int остается неименованный промежуток.

Структуру полей можно заполнить неименованными промежутками с применением ширин неименованных полей. Использование неименованного поля шириной 0 приводит к тому, что следующее поле выравнивается по следующей области целочисленного значения:

```
struct {
   unsigned int field1 : 1;
   unsigned int       : 2;
   unsigned int field2 : 1;
   unsigned int       : 0;
   unsigned int field3 : 1;
} stuff;
```

Здесь между полями stuff.field1 и stuff.field2 имеется 2-битовый промежуток, а поле stuff.field3 хранится в следующей области int.

Важной зависимостью от системы является порядок, в котором поля помещаются в область int. В одних системах поддерживается порядок слева направо, в других — справа налево. Кроме того, системы различаются местоположением границ между полями. По этим причинам битовые поля не особенно переносимы. Однако обычно они применяются в целях, не предполагающих переносимость, таких как размещение данных в точной форме, используемой отдельным аппаратным устройством.

Пример с битовыми полями

Битовые поля часто применяются в качестве более компактного способа хранения данных. Предположим, что вы решили представить свойства выводимого на экран окна. Давайте отложим в сторону все сложности графики и предположим, что окно обладает только перечисленными ниже свойствами.

- Окно может быть прозрачным или непрозрачным.
- Цвет фона выбирается из следующей палитры: черный, красный, зеленый, желтый, синий, пурпурный, голубой и белый.
- Рамка может быть скрыта или отображена.
- Цвет рамки выбирается из той же палитры, что и цвет фона.
- Для рамки применяются три стиля линии: сплошная, пунктирная и штриховая.

Для каждого свойства можно было бы использовать отдельную переменную или полноразмерный член структуры, но это привело бы к напрасному расходу битов. Например, для указания прозрачности или непрозрачности окна достаточно одного бита. То же самое можно сказать о свойстве отображения или сокрытия рамки. Восемь возможных значений цвета могут быть представлены 3-битовым элементом, а 2-битового элемента более чем достаточно для представления трех возможных стилей рамки. Таким образом, для представления всех пяти свойств достаточно 10 битов.

Один из вариантов представления информации предусматривает применение заполнителей, чтобы поместить связанную с фоном окна информацию в один байт, а связанную с рамкой — во второй. Это реализовано в следующем объявлении struct box_props:

```
struct box_props {
    bool opaque                : 1;
    unsigned int fill_color    : 3;
    unsigned int               : 4;
    bool show_border           : 1;
    unsigned int border_color  : 3;
    unsigned int border_style  : 2;
    unsigned int               : 2;
};
```

В результате использования заполнителей размер структуры увеличивается до 16 битов. Без них было бы достаточно 10 битов. Однако имейте в виду, что в C для структур с битовыми полями в качестве базовой единицы размещения применяется тип unsigned int. Поэтому, даже если структура содержит единственный элемент, которым является однобитовое поле, структура будет иметь такой же размер, как у типа unsigned int, что в нашей системе составляет 32 бита. Кроме того, в этом коде предполагается, что тип _Bool из C99 доступен и в заголовочном файле stdbool.h ему назначен псевдоним bool.

Для члена opaque можно использовать значение 1 для указания непрозрачности окна и значение 0 — для прозрачности. То же самое применимо к члену show_border.

Для цветов можно использовать простое представление RGB (красный, зеленый, синий). Это основные цвета для смешивания спектра. В мониторе для воспроизведения различных цветов применяется смешанное свечение красных, зеленых и синих пикселей. В ранних моделях мониторов каждый пиксель мог иметь только включенное или выключенное состояние, поэтому для представления интенсивности каждой из трех составляющих было достаточно одного бита. Обычно левый бит представлял интенсивность синего, средний – интенсивность зеленого, а правый – красного цвета. В табл. 15.3 показаны восемь возможных комбинаций. Они могут служить значениями для членов `fill_color` и `border_color`. Наконец, значения 0, 1 и 2 могут представлять сплошной, пунктирный и штриховой тип линий, определяемый членом `border_style`.

Таблица 15.3. Простое представление цветов

Комбинация битов	Десятичный эквивалент	Цвет
000	0	Черный
001	1	Красный
010	2	Зеленый
011	3	Желтый
100	4	Синий
101	5	Пурпурный
110	6	Голубой
111	7	Белый

В листинге 15.3 структура `box_props` используется в простом примере. Директивы `#define` применяются для создания символических констант, представляющих возможные значения членов. Обратите внимание, что основные цвета представлены включением единственного бита. Остальные цвета могут представляться комбинациями основных цветов. Например, пурпурный цвет создается включением битов синего и красного цветов, поэтому его можно записать как комбинацию `BLUE | RED`.

Листинг 15.3. Программа `fields.c`

```
/* fields.c -- определение и использование полей */
#include <stdio.h>
#include <stdbool.h>   // C99, определение bool, true, false
/* стили линии    */
#define SOLID   0
#define DOTTED  1
#define DASHED  2
/* основные цвета  */
#define BLUE    4
#define GREEN   2
#define RED     1
/* смешанные цвета */
#define BLACK   0
#define YELLOW  (RED | GREEN)
#define MAGENTA (RED | BLUE)
#define CYAN    (GREEN | BLUE)
#define WHITE   (RED | GREEN | BLUE)
const char * colors[8] = {"черный", "красный", "зеленый", "желтый",
    "синий", "пурпурный", "голубой", "белый"};
```

```
struct box_props {
   bool opaque               : 1;      // или unsigned int (до C99)
   unsigned int fill_color   : 3;
   unsigned int              : 4;
   bool show_border          : 1;      // или unsigned int (до C99)
   unsigned int border_color : 3;
   unsigned int border_style : 2;
   unsigned int              : 2;
};
void show_settings(const struct box_props * pb);
int main(void)
{
   /* создание и инициализация структуры box_props */
   struct box_props box = {true, YELLOW , true, GREEN, DASHED};

   printf("Исходные настройки окна:\n");
   show_settings(&box);

   box.opaque = false;
   box.fill_color = WHITE;
   box.border_color = MAGENTA;
   box.border_style = SOLID;
   printf("\nИзмененные настройки окна:\n");
   show_settings(&box);

   return 0;
}
void show_settings(const struct box_props * pb)
{
   printf("Окно %s.\n",
          pb->opaque == true ? "непрозрачно": "прозрачно");
   printf("Цвет фона %s.\n", colors[pb->fill_color]);
   printf("Рамка %s.\n",
          pb->show_border == true ? "отображается" : "не отображается");
   printf("Цвет рамки %s.\n", colors[pb->border_color]);
   printf ("Стиль рамки ");
   switch(pb->border_style)
   {
      case SOLID  : printf("сплошной.\n"); break;
      case DOTTED : printf("пунктирный.\n"); break;
      case DASHED : printf("штриховой.\n"); break;
      default     : printf("неизвестного типа.\n");
   }
}
```

Вот вывод, полученный из программы:

```
Исходные настройки окна:
Окно непрозрачно.
Цвет фона желтый.
Рамка отображается.
Цвет рамки зеленый.
Стиль рамки штриховой.

Измененные настройки окна:
Окно прозрачно.
Цвет фона белый.
Рамка отображается.
Цвет рамки пурпурный.
Стиль рамки сплошной.
```

Отметим несколько моментов. Прежде всего, структуру битовых полей можно инициализировать с использованием обычного для структур синтаксиса:

```
struct box_props box = {YES, YELLOW , YES, GREEN, DASHED};
```

Аналогично можно присваивать значения элементам битовых полей:

```
box.fill_color = WHITE;
```

Кроме того, член битового поля может служить выражением в операторе `switch`. Он даже может выступать в качестве индекса массива:

```
printf("Цвет фона %s.\n", colors[pb->fill_color]);
```

Обратите внимание, что массив `colors` был определен так, чтобы каждое значение индекса соответствовало строковому представлению названия цвета, имеющего значение индекса, которое совпадает с числовым значением цвета. Например, индекс `1` соответствует строке `"красный"` и константа `RED` имеет значение `1`.

Битовые поля и побитовые операции

Битовые поля и побитовые операции — это два альтернативных подхода к решению задачи программирования одного и того же типа. Это значит, что часто можно применять любой из подходов. В предыдущем примере для хранения информации о графическом окне использовалась структура с размером, как у типа `unsigned int`. Ту же самую информацию можно было бы сохранить в переменной типа `unsigned int`. Затем вместо синтаксиса членства в структуре можно было бы применять побитовые операции. Обычно такая методика не очень удобна. Рассмотрим пример, в котором задействованы оба подхода. (Оба подхода здесь применяются для иллюстрации отличий между ними, а вовсе не из-за того, чтобы внушить мысль о целесообразности их одновременного использования!)

В качестве средства комбинирования подхода на основе структуры и подхода на базе побитовых операций можно воспользоваться объединением. Исходя из существующего объявления типа `struct box_props`, можно объявить следующее объединение:

```
union Views    /* взгляд на данные как на struct или как на unsigned short */
{
   struct box_props st_view;
   unsigned short us_view;
};
```

В некоторых системах переменная `unsigned short` и структура `box_props` занимают 16 битов в памяти. В других системах, таких как наша, `unsigned short` и `box_props` занимают 32 бита. В любом случае это объединение позволяет применять член `st_view`, чтобы трактовать отведенную память как структуру, или использовать член `us_view`, чтобы рассматривать тот же самый блок памяти как значение `unsigned short`. Какие битовые поля структуры соответствуют отдельным битам переменной типа `unsigned short`? Это зависит от реализации и оборудования. В следующем примере предполагается, что структуры загружены в память, начиная с младших битов и заканчивая старшими битами байта. Другими словами, первое битовое поле в структуре соответствует биту 0 слова. (Для простоты эта идея иллюстрируется на рис. 15.3 для 16-битового единицы.)

В листинге 15.4 объединение `Views` применяется для сравнения подходов на основе битовых полей и побитовых операций. Здесь `box` — это объединение `Views`, поэтому `box.st_view` представляет собой структуру `box_props`, использующую битовые поля, а `box.ui_view` — те же самые данные, но представленные как значение `unsigned short`.

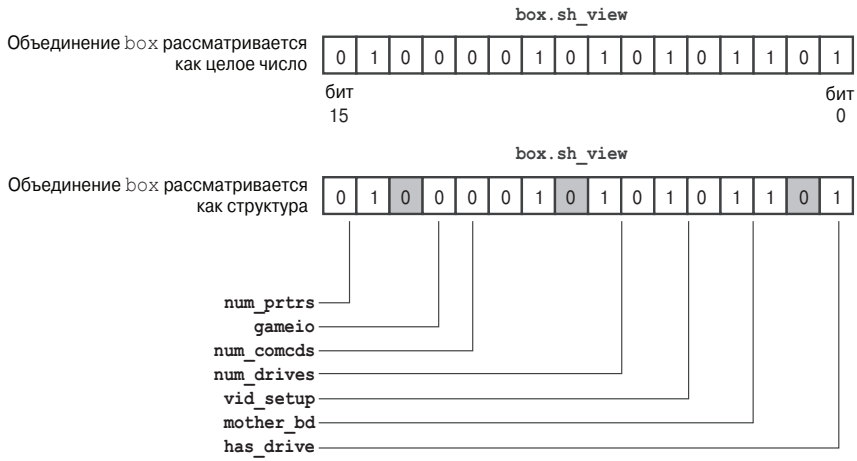

Рис. 15.3. *Объединение как целое число и как структура*

Вспомните, что объединение может иметь инициализированный первый член, поэтому установленные значения соответствуют представлению структуры. Программа отображает свойства окна с помощью функции, основанной на представлении структуры, и также посредством функции, основанной на представлении `unsigned short`. Любой из подходов обеспечивает доступ к данным, но по-разному. Вдобавок в программе применяется определенная ранее в главе функция `itobs()`, которая позволяет отобразить данные в виде строки двоичных цифр, чтобы можно было видеть, какие биты включены, а какие выключены.

Листинг 15.4. Программа `dualview.c`

```
/* dualview.c -- битовые поля и побитовые операции */
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
/* КОНСТАНТЫ БИТОВЫХ ПОЛЕЙ */
/* стили линии    */
#define SOLID  0
#define DOTTED 1
#define DASHED 2
/* основные цвета  */
#define BLUE   4
#define GREEN  2
#define RED    1
/* смешанные цвета */
#define BLACK   0
#define YELLOW  (RED | GREEN)
#define MAGENTA (RED | BLUE)
#define CYAN    (GREEN | BLUE)
#define WHITE   (RED | GREEN | BLUE)

/* ПОБИТОВЫЕ КОНСТАНТЫ   */
#define OPAQUE           0x1
#define FILL_BLUE        0x8
#define FILL_GREEN       0x4
```

```
#define FILL_RED         0x2
#define FILL_MASK        0xE
#define BORDER           0x100
#define BORDER_BLUE      0x800
#define BORDER_GREEN     0x400
#define BORDER_RED       0x200
#define BORDER_MASK      0xE00
#define B_SOLID          0
#define B_DOTTED         0x1000
#define B_DASHED         0x2000
#define STYLE_MASK       0x3000

const char * colors[8] = {"черный", "красный", "зеленый", "желтый",
   "синий", "пурпурный", "голубой", "белый"};
struct box_props {
   bool opaque               : 1;
   unsigned int fill_color   : 3;
   unsigned int              : 4;
   bool show_border          : 1;
   unsigned int border_color : 3;
   unsigned int border_style : 2;
   unsigned int              : 2;
};

union Views    /* взгляд на данные как на struct или как на unsigned short */
{
   struct box_props st_view;
   unsigned short us_view;
};

void show_settings(const struct box_props * pb);
void show_settings1(unsigned short);
char * itobs(int n, char * ps);

int main(void)
{
   /* создание объекта Views, инициализация представления в виде структуры */
   union Views box = {{true, YELLOW , true, GREEN, DASHED}};
   char bin_str[8 * sizeof(unsigned int) + 1];

   printf("Исходные настройки окна:\n");
   show_settings(&box.st_view);
   printf("\nНастройки окна с использованием представления unsigned short:\n");
   show_settings1(box.us_view);

   printf("комбинация битов %s\n",
         itobs(box.us_view,bin_str));
   box.us_view &= ~FILL_MASK;               /* очистить биты фона */
   box.us_view |= (FILL_BLUE | FILL_GREEN); /* переустановить фон */
   box.us_view ^= OPAQUE;                   /* переключить прозрачность */
   box.us_view |= BORDER_RED;               /* ошибочный подход */
   box.us_view &= ~STYLE_MASK;              /* очистить биты стиля */
   box.us_view |= B_DOTTED;                 /* установить пунктирный стиль */
   printf("\nИзмененные настройки окна:\n");
   show_settings(&box.st_view);
   printf("\nНастройки окна с использованием представления unsigned short:\n");
   show_settings1(box.us_view);
   printf("Комбинация битов %s\n",
         itobs(box.us_view,bin_str));

   return 0;
}
```

```
void show_settings(const struct box_props * pb)
{
   printf("Окно %s.\n",
           pb->opaque == true ? "непрозрачно": "прозрачно");
   printf("Цвет фона %s.\n", colors[pb->fill_color]);
   printf("Рамка %s.\n",
           pb->show_border == true ? "отображается" : "не отображается");
   printf("Цвет рамки %s.\n", colors[pb->border_color]);
   printf("Стиль рамки ");
   switch(pb->border_style)
   {
      case SOLID  : printf("сплошной.\n"); break;
      case DOTTED : printf("пунктирный.\n"); break;
      case DASHED : printf("штриховой.\n"); break;
      default     : printf("неизвестного типа.\n");
   }
}
void show_settings1(unsigned short us)
{
   printf("Окно %s.\n",
         (us & OPAQUE) == OPAQUE? "непрозрачно": "прозрачно");
   printf("Цвет фона %s.\n",
          colors[(us >> 1) & 07]);
   printf("Рамка %s.\n",
         (us & BORDER) == BORDER? "отображается" : "не отображается");
   printf ("Стиль рамки ");
   switch(us & STYLE_MASK)
   {
      case B_SOLID  : printf("сплошной.\n"); break;
      case B_DOTTED : printf("пунктирный.\n"); break;
      case B_DASHED : printf("штриховой.\n"); break;
      default       : printf("неизвестного типа.\n");
   }
   printf("Цвет рамки %s.\n",
         colors[(us >> 9) & 07]);
}
char * itobs(int n, char * ps)
{
   int i;
   const static int size = CHAR_BIT * sizeof(int);
   for (i = size - 1; i >= 0; i--, n >>= 1)
      ps[i] = (01 & n) + '0';
   ps[size] = '\0';
   return ps;
}
```

Ниже приведен вывод.

```
Исходные настройки окна:
Окно непрозрачно.
Цвет фона желтый.
Рамка отображается.
Цвет рамки зеленый.
Стиль рамки штриховой.
```

```
Настройки окна с использованием представления unsigned short:
Окно непрозрачно.
Цвет фона желтый.
Рамка отображается.
Стиль рамки штриховой.
Цвет рамки зеленый.
Комбинация битов 00000000000000000010010100000111

Измененные настройки окна:
Окно прозрачно.
Цвет фона голубой.
Рамка отображается.
Цвет рамки желтый.
Стиль рамки пунктирный.

Настройки окна с использованием представления unsigned short:
Окно прозрачно.
Цвет фона голубой.
Рамка не отображается.
Стиль рамки пунктирный.
Цвет рамки желтый.
Комбинация битов 00000000000000000001011100001100
```

В коде есть несколько моментов, которые необходимо обсудить. Одно из отличий между этими двумя представлениями состоит в том, что побитовому представлению нужна информация о позициях. Например, для представления синего цвета в программе используется константа `BLUE`, имеющая числовое значение `4`. Но поскольку данные организованы в структуре, на самом деле хранить настройку синего цвета для фона будет бит 3 (не забывайте, что нумерация начинается с нуля (см. рис. 15.1)), а настройку синего цвета для рамки — бит 11. Таким образом, в программе определен ряд новых констант:

```
#define FILL_BLUE      0x8
#define BORDER_BLUE    0x800
```

Здесь `0x8` — это значение, когда в 1 установлен только бит 3, а `0x800` — значение, когда в 1 установлен только бит 11. Первую константу можно применять при установке бита синего цвета для фона окна, а вторую — при установке бита синего цвета для рамки. Шестнадцатеричная запись упрощает выяснение, какие биты задействованы. Вспомните, что каждая шестнадцатеричная цифра представляет четыре бита. Следовательно, значение `0x800` соответствует комбинации битов `0x8`, но с дописанными восемью битами с состоянием 0. Глядя на десятичные эквиваленты, 2048 и 8, заметить такую связь гораздо труднее.

Если значения являются степенями 2, можно воспользоваться операцией сдвига влево. Например, последние две директивы `#define` можно заменить следующим образом:

```
#define FILL_BLUE      1<<3
#define BORDER_BLUE    1<<11
```

Во втором операнде указана степень для возведения числа 2. Так, значение `0x8` равно 2^3, а значение `0x800` — 2^{11}. Аналогично, выражение `1<<n` дает целочисленное значение, у которого в 1 установлен только бит `n`. Выражения наподобие `1<<11` являются константными и вычисляются на этапе компиляции.

Вместо директивы `#define` для создания символических констант можно применять перечисление.

Например, можно поступить так:

```
enum { OPAQUE = 0x1, FILL_BLUE = 0x8, FILL_GREEN = 0x4, FILL_RED = 0x2,
       FILL_MASK = 0xE, BORDER = 0x100, BORDER_BLUE = 0x800,
       BORDER_GREEN = 0x400, BORDER_RED = 0x200, BORDER_MASK = 0xE00,
       B_DOTTED = 0x1000, B_DASHED = 0x2000, STYLE_MASK = 0x3000 };
```

Если вы не намерены создавать переменные с типом этого перечисления, указывать имя в объявлении не обязательно.

Обратите внимание, что использовать побитовые операции для изменения настроек сложнее. В качестве примера давайте установим голубой цвет для фона окна. В данном случае недостаточно просто включить биты, соответствующие синему и зеленому цветам:

```
box.us_view |= (FILL_BLUE | FILL_GREEN);     /* сбросить фон */
```

Дело в том, что цвет также полагается на настройку бита, отвечающего за красный цвет. Если этот бит был включен ранее (скажем, для получения желтого цвета), то приведенный код оставит бит красного цвета установленным и установит в 1 биты синего и зеленого цветов, давая в результате белый цвет. Обойти эту проблему проще всего, сначала отключив все биты, отвечающие за цвет, и лишь затем устанавливать новые значения. Именно поэтому в программе содержится следующий код:

```
box.us_view &= ~FILL_MASK;                  /* очистить биты фона */
box.us_view |= (FILL_BLUE | FILL_GREEN);    /* переустановить фон */
```

Для демонстрации того, что может произойти, если предварительно не очистить соответствующие биты, в программе также предусмотрена такая строка:

```
box.us_view |= BORDER_RED;                  /* ошибочный подход */
```

Из-за того, что бит `BORDER_GREEN` уже был установлен, результирующим цветом будет `BORDER_GREEN | BORDER_RED`, что соответствует желтому цвету.

В ситуациях подобного рода применять битовые поля проще:

```
box.st_view.fill_color = CYAN;              /* эквивалент с битовым полем */
```

Тогда нет нужды в предварительной очистке битов. Кроме того, члены битовых полей допускают использование одних и тех же значений цвета для рамки и фона окна. В случае подхода с побитовыми операциями придется применять отличающиеся значения (значения, отражающие действительные позиции битов).

Далее, сравните следующие два оператора вывода:

```
printf("Цвет рамки %s.\n", colors[pb->border_color]);
printf("Цвет рамки %s.\n", colors[(us >> 9) & 07]);
```

В первом операторе выражение `pb->border_color` имеет значение из диапазона 0–7, поэтому его можно использовать как индекс в массиве `colors`. Получить ту же информацию с помощью побитовых операций сложнее. Один из подходов предусматривает применение `ui >> 9` для сдвига битов цвета рамки в самую правую позицию (биты 0–2) с последующим объединением полученного значения с маской `07`, в результате чего все биты кроме трех самых правых будут отключены. То, что осталось, будет находиться в диапазоне 0–7 и может использоваться в качестве индекса для массива `colors`.

Внимание!

Соответствие между битовыми полями и позициями битов зависит от реализации. Например, при выполнении программы из листинга 15.4 в старой системе Macintosh PowerPC получается следующий вывод:

```
Исходные настройки окна:
Окно непрозрачно.
Цвет фона желтый.
Рамка отображается.
Цвет рамки зеленый.
Стиль рамки штриховой.

Настройки окна с использованием представления unsigned short:
Окно прозрачно.
Цвет фона черный.
Рамка не отображается.
Стиль рамки сплошной.
Цвет рамки черный.
Комбинация битов 10110000101010000000000000000000

Измененные настройки окна:
Окно прозрачно.
Цвет фона желтый.
Рамка отображается.
Цвет рамки зеленый.
Стиль рамки штриховой.

Настройки окна с использованием представления unsigned short:
Окно прозрачно.
Цвет фона голубой.
Рамка отображается.
Стиль рамки пунктирный.
Цвет рамки красный.
Комбинация битов 10110000101010000001001000001101
```

Здесь изменения затрагивают те же биты, что и ранее, но в системе Macintosh PowerPC загрузка структур в память осуществляется по-другому. В частности, первое битовое поле загружается, начиная со старшего, а не младшего бита. Поэтому представление структуры касается первых 16 битов (которые следуют в порядке, отличающемся от версии IBM PC), тогда как представление `unsigned int` затрагивает последние 16 битов. Таким образом, допущения, сделанные относительно позиций битов в листинге 15.4, для Macintosh PowerPC некорректны, а побитовые операции, применяемые для изменения настроек прозрачности и цвета фона, изменяют не те биты.

Средства выравнивания (C11)

Средства выравнивания C11 по своей природе больше ориентированы на манипулирование байтами, чем битами, но они также отражают возможность языка C иметь дело с оборудованием. В этом контексте выравнивание относится к тому, как объекты располагаются в памяти. Например, для максимальной эффективности система может требовать, чтобы значение типа `double` хранилось в памяти по адресу, кратному 4, но разрешать значению типа `char` храниться по любому адресу. Большинству программистов редко когда придется заботиться о выравнивании. Но в некоторых ситуациях контроль над выравниванием позволяет извлечь выгоду, например, при передаче данных из одного физического места в другое либо при вызове инструкций, которые оперируют на множестве элементов данных одновременно.

Операция _Alignof выдает требования к выравниванию указанного типа. Для ее использования необходимо после ключевого слова _Alignof поместить имя типа в круглых скобках:

```
size_t d_align = _Alignof(float);
```

Полученное значение, скажем, 4 для d_align, говорит о том, что объекты float имеют требование к выравниванию, соответствующее 4. Это означает, что 4 является количеством байтов между следующими друг за другом адресами для хранения значений упомянутого типа. В общем случае значения выравнивания должны быть неотрицательными целыми числами, которые представляют собой степень 2. Более высокие значения выравнивания считаются *более жесткими* или *более строгими*, чем меньшие значения, в то время как меньшие значения трактуются как *более слабые*.

С помощью спецификатора _Alignas можно запрашивать конкретное выравнивание для переменной или типа. Однако вы не должны запрашивать выравнивание, которое слабее фундаментального выравнивания, принятого для типа. Например, если требование к выравниванию для float составляет 4, не запрашивайте значение выравнивания, равное 1 или 2. Этот спецификатор применяется как часть объявления, и за ним следует пара круглых скобок, содержащая либо значение выравнивания, либо тип:

```
_Alignas(double) char c1;
_Alignas(8) char c2;
unsigned char _Alignas(long double) c_arr[sizeof(long double)];
```

На заметку!

На момент написания книги компилятор Clang (версии 3.2) требовал, чтобы спецификатор _Alignas(*тип*) располагался после спецификатора типа, как в третьей строка приведенного выше кода. Тем не менее, компилятор GCC 4.7.3 распознает оба порядка следования, как и последующая версия (3.3) компилятора Clang.

В листинге 15.5 показан короткий пример использования _Alignas и _Alignof.

Листинг 15.5. Программа `align.c`

```
//  align.c -- использование _Alignof и _Alignas (C11)
#include <stdio.h>
int main(void)
{
   double dx;
   char ca;
   char cx;
   double dz;
   char cb;
   char _Alignas(double) cz;

   printf("Выравнивание char:   %zd\n", _Alignof(char));
   printf("Выравнивание double: %zd\n", _Alignof(double));
   printf("&dx: %p\n", &dx);
   printf("&ca: %p\n", &ca);
   printf("&cx: %p\n", &cx);
   printf("&dz: %p\n", &dz);
   printf("&cb: %p\n", &cb);
   printf("&cz: %p\n", &cz);

   return 0;
}
```

Вот пример вывода:

```
Выравнивание char:   1
Выравнивание double: 8
&dx: 0x7fff5fbff660
&ca: 0x7fff5fbff65f
&cx: 0x7fff5fbff65e
&dz: 0x7fff5fbff650
&cb: 0x7fff5fbff64f
&cz: 0x7fff5fbff648
```

В нашей системе значение выравнивания 8 для типа `double` подразумевает, что значения этого типа сохраняются по адресам, кратным 8. Шестнадцатеричные адреса, заканчивающиеся на 0 или 8, являются кратными 8, и адреса такого вида применялись для двух переменных `double`, а также переменной `char` по имени `cz`, которой было назначено значение выравнивания для типа `double`. Поскольку значением выравнивания для `char` было 1, компилятор мог использовать для переменных этого типа любые адреса.

Включение заголовочного файла `stdalign.h` позволяет применять псевдонимы `alignas` и `alignof` для `_Alignas` и `_Alignof`. Они соответствуют ключевым словам в C++. В C11 также появилась возможность выравнивания для выделенной памяти за счет добавления в библиотеку `stdlib.h` новой функции распределения памяти со следующим прототипом:

```
void *aligned_alloc(size_t alignment, size_t size);
```

В первом параметре указывается требуемое выравнивание, а во втором — количество необходимых байтов, которое должно быть кратным значению первого параметра. Как и в случае других функций распределения памяти, по завершении работы с выделенной памятью используйте функцию `free()`, чтобы ее освободить.

Ключевые понятия

Одной из особенностей, которая отличает C от большинства языков высокого уровня, является возможность доступа к отдельным битам целого числа. Это часто играет ключевую роль при взаимодействии с аппаратными устройствами и операционными системами.

Язык C обладает двумя основными средствами для работы с битами. Первое — это семейство побитовых операций, а второе — создание битовых полей в структуре.

В C11 добавлена возможность контроля требования к выравниванию памяти и запрашивания более строгих таких требований.

Обычно, хотя и не всегда, программы, в которых задействованы эти средства, привязаны к конкретным аппаратным платформам или операционным системам и не задуманы быть переносимыми.

Резюме

Вычислительные системы тесно связаны с двоичной системой счисления, поскольку нули и единицы могут представлять выключенное и включенное состояние битов памяти и регистров. Хотя язык C не разрешает записывать целые числа в двоичной форме, он распознает восьмеричные и шестнадцатеричные системы записи. Подобно тому, как двоичная цифра представляет один бит, восьмеричная цифра представляет три бита, а шестнадцатеричная — четыре бита. Такая взаимосвязь позволяет сравнительно просто преобразовывать двоичные числа в восьмеричную или шестнадцатеричную форму.

В языке C предлагается несколько побитовых операций, которые так называются из-за того, что воздействуют на каждый бит значения независимым образом. Побитовая операция отрицания (~) инвертирует каждый бит своего операнда, преобразуя 1 в 0 и наоборот. Побитовая операция "И" (&) формирует значение из двух операндов. Каждый бит в значении устанавливается в 1, если соответствующие биты в обоих операндах равны 1; в противном случае бит устанавливается в 0. Побитовая операция "ИЛИ" (|) также формирует значение из двух операндов. Каждый бит в значении устанавливается в 1, если соответствующий бит в одном из двух или в обоих операндах равен 1; в противном случае бит устанавливается в 0. Побитовая операция исключающего "ИЛИ" (^) действует аналогично с тем отличием, что результирующий бит устанавливается в 1, только если соответствующий бит равен 1 в одном или в другом операнде, но не в обоих.

Вдобавок в C имеются операции сдвига влево (<<) и вправо (>>). Каждая из них создает значение, формируемое путем сдвига битов (влево или вправо) левого операнда на количество позиций, указанное в правом операнде. При операции сдвига влево освобождаемые биты устанавливаются в 0. При операции сдвига вправо освобождаемые биты устанавливаются в 0 для значений без знака. Для значений со знаком поведение операции сдвига вправо зависит от реализации.

Для обращения к отдельным битам или к группе битов в значении можно применять битовые поля. Детали такого манипулирования зависят от реализации.

С помощью операции `_Alignas` можно устанавливать требования к выравниванию при сохранении данных. Инструменты для работы с битами помогают программам на C взаимодействовать с оборудованием, поэтому они чаще всего привязаны к контексту конкретной реализации.

Вопросы для самоконтроля

Ответы на вопросы для самоконтроля приведены в приложении A.

1. Преобразуйте следующие десятичные значения в двоичную форму:
   а. 3
   б. 13
   в. 59
   г. 119
2. Преобразуйте следующие двоичные значения в десятичную, восьмеричную и шестнадцатеричную форму:
   а. `00010101`
   б. `01010101`
   в. `01001100`
   г. `10011101`
3. Вычислите следующие выражения; предположите, что каждое значение имеет 8 битов:
   а. `~3`
   б. `3 & 6`
   в. `3 | 6`
   г. `1 | 6`
   д. `3 ^ 6`
   е. `7 >> 1`
   ж. `7 << 2`

4. Вычислите следующие выражения; предположите, что каждое значение имеет 8 битов:
   а. `~0`
   б. `!0`
   в. `2 & 4`
   г. `2 && 4`
   д. `2 | 4`
   е. `2 || 4`
   ж. `5 << 3`

5. Поскольку в ASCII-коде используются только последние 7 битов, иногда желательно маскировать остальные биты. Как будет выглядеть подходящая маска в двоичной форме? В десятичной? В восьмеричной? В шестнадцатеричной?

6. В листинге 15.2 следующий код

```
while (bits-- > 0)
{
  mask |= bitval;
  bitval <<= 1;
}
```

   можно заменить таким фрагментом:

```
while (bits-- > 0)
{
  mask += bitval;
  bitval *= 2;
}
```

   и программа по-прежнему будет работать. Означает ли это, что действие `*= 2` эквивалентно `<<= 1`? А как насчет `|=` и `+=`?

7. а. Компьютер Tinkerbell содержит в специальном байте информацию, касающуюся оборудования. Этот байт может быть прочитан программой, и он содержит следующую информацию:

Биты	Описание
0–1	Количество дисководов 1.44 Мбайт
2	Не используется
3–4	Количество приводов чтения компакт-дисков
5	Не используется
6–7	Количество жестких дисков

   Подобно IBM PC, компьютер Tinkerbell заполняет битовые поля структуры справа налево. Создайте шаблон битовых полей, подходящий для хранения информации.

   б. Компьютер Klinkerbell, ближайший клон Tinkerbell, заполняет битовые поля структур слева направо. Создайте соответствующий шаблон битовых полей для системы Klinkerbell.

Упражнения по программированию

1. Напишите функцию, которая преобразует строку с двоичным представлением в числовое значение. Другими словами, если есть

   ```
   char * pbin = "01001001";
   ```

   то переменную `pbin` можно передать этой функции в качестве аргумента, и функция должна возвратить значение `25` типа `int`.

2. Напишите программу, которая читает две строки с двоичным представлением как аргументы командной строки и выводит результаты применения операции `~` к каждому числу, а также результаты применения операций `&`, `|` и `^` к паре чисел. Отобразите результаты в виде двоичных строк. (Если среда командной строки недоступна, обеспечьте в программе интерактивный ввод строк.)

3. Напишите функцию, которая принимает аргумент типа `int` и возвращает количество включенных битов в нем. Протестируйте функцию в какой-нибудь программе.

4. Напишите функцию, которая принимает два аргумента типа `int`: значение и позицию бита. Функция должна возвращать 1, если бит в этой позиции равен 1, и 0 в противном случае. Протестируйте функцию в какой-нибудь программе.

5. Напишите функцию, которая циклически сдвигает биты значения типа `unsigned int` на указанное количество позиций влево. Например, функция `rotate_l(x,4)` перемещает биты значения `x` на четыре позиции влево, при этом утраченные слева биты воспроизводятся в правой части комбинации. Другими словами, вытесненный старший бит помещается в позицию младшего бита. Протестируйте функцию в какой-нибудь программе.

6. Разработайте структуру битовых полей, которая содержит следующую информацию:
   - Идентификатор шрифта: число от 0 до 255
   - Размер шрифта: число от 0 до 127
   - Выравнивание: число от 0 до 2, представляющее опции выравнивания влево, по центру и вправо
   - Полужирный: отключен (0) или включен (1)
   - Курсив: отключен (0) или включен (1)
   - Подчеркнутый: отключен (0) или включен (1)

   Используйте эту структуру в программе, которая отображает параметры шрифта и дает пользователю возможность менять параметры с помощью циклического меню. Ниже приводится пример выполнения программы:

```
ИД    РАЗМЕР    ВЫРАВНИВАНИЕ      Ж       К       Ч
 1       12       влево          откл.   откл.   откл.

ш) изменить шрифт    р) изменить размер     в) изменить выравнивание
ж) полужирный        к) курсив              п) подчеркнутый
з) завершить
р
```

```
Введите размер шрифта (0-127): 36
ИД    РАЗМЕР    ВЫРАВНИВАНИЕ      Ж       К       Ч
 1       36      влево          откл.    откл.    откл.

ш) изменить шрифт    р) изменить размер     в) изменить выравнивание
ж) полужирный        к) курсив              п) подчеркнутый
з) завершить
в
Выберите выравнивание:
л) влево             ц) по центру           п) вправо
п

ИД    РАЗМЕР    ВЫРАВНИВАНИЕ      Ж       К       Ч
 1       36      вправо         откл.    откл.    откл.

ш) изменить шрифт    р) изменить размер     в) изменить выравнивание
ж) полужирный        к) курсив              п) подчеркнутый
з) завершить
к

ИД    РАЗМЕР    ВЫРАВНИВАНИЕ      Ж       К       Ч
 1       36      вправо         откл.    вкл.     откл.

ш) изменить шрифт    р) изменить размер     в) изменить выравнивание
ж) полужирный        к) курсив              п) подчеркнутый
з) завершить
з
Программа завершена.
```

Чтобы обеспечить преобразование вводимых значений идентификатора и размера шрифта в значения из указанного диапазона, программа должна применять операцию `&` и подходящие маски.

7. Напишите программу с таким же поведением, как в упражнении 6, но используйте для хранения информации о шрифте переменную типа `unsigned long`, а для манипулирования этой информацией — побитовые операции вместо членов структуры с битовыми полями.