KAPITEL 11

Bit-Operationen

Sein oder nicht sein, das ist hier die Frage.
– Shakespeare über Boolesche Algebra

In diesem Kapitel besprechen wir bitorientierte Operationen. Ein Bit ist die kleinste Informationseinheit; normalerweise wird sie durch die Werte 1 und 0 repräsentiert. (Weitere Repräsentationen sind an/aus (on/off), wahr/falsch (true/false) und ja/nein (yes/no).) Bitmanipulationen dienen der Maschinensteuerung auf niedrigster Ebene. Der Programmierer kann damit »unter die Motorhaube« des Rechners gelangen. Viele Applikationen werden nie irgendwelche Bit-Operationen benötigen. Aber in Lowlevel-Programmen wie Gerätetreibern oder Grafikprogrammen auf Pixelebene kommt man ohne Bit-Operationen nicht aus.

Acht Bits bilden ein Byte, das in C++ durch den Datentyp **char** repräsentiert wird. Ein Byte könnte die folgende Bitfolge enthalten: 01100100.

Die Binärzahl 01100100 kann auch als hexadezimale Zahl 0x64 geschrieben werden (C++ verwendet das Präfix »0x«, um hexadezimale Zahlen, also Zahlen zur Basis 16, anzugeben.) Die hexadezimale Notation ist bequem für die Repräsentation binärer Daten, weil jede hexadezimale Ziffer vier binäre Bits repräsentiert. Tabelle 11-1 gibt die Konvertierung von hexadezimal (hex) nach binär an.

Die hexadezimale Zahl 0xAF repräsentiert also die binäre Zahl 10101111.

Tabelle 11-1: Hexadezimale und binäre Zahlen

Hex	Binär	Hex	Binär
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Bit-Operatoren

Bit- oder *bitweise Operatoren* ermöglichen es dem Programmierer, mit einzelnen Bits zu arbeiten. Beispielsweise enthält ein **short int** (auf den meisten Rechnern) 16 Bits. Die Bit-Operatoren behandeln alle diese als unabhängige Bits. Im Gegensatz dazu behandelt ein Additionsoperator die 16 Bits als eine einzige 16-Bit-Zahl.

Mit Bit-Operatoren können Sie Bits setzen, löschen, abfragen und andere Operationen darauf durchführen. Die vorhandenen Bit-Operatoren sind in Tabelle 11-2 aufgeführt.

Tabelle 11-2: Bit-Operatoren

Operator	Bedeutung
&	Bitweises UND
|	Bitweises ODER
^	Bitweises exklusives ODER
~	Komplement
<<	Nach links verschieben
>>	Nach rechts verschieben

Diese Operatoren funktionieren auf allen Integer- und Zeichen-Datentypen.

Der UND-Operator (&)

Der UND-*Operator* vergleicht zwei Bits. Wenn beide 1 sind, ist auch das Ergebnis 1. Die Ergebnisse des UND-Operators sind in Tabelle 11-3 definiert.

Tabelle 11-3: Der UND-Operator

Bit1	Bit2	Bit1 & Bit2
0	0	0
0	1	0
1	0	0
1	1	1

Wenn zwei 8-Bit-Variablen (`char`-Variablen) mit UND verknüpft werden, dann arbeitet der UND-Operator auf jedem Bit einzeln. Der folgende Programmausschnitt illustriert diese Operation. (In der darin enthaltenen Ausgabeanweisung teilt `hex` dem System mit, Zahlen in hexadezimalem Format auszugeben, und mit `dec` wird wieder zu Dezimalzahlen zurückgekehrt. Nähere Informationen dazu finden Sie in Kapitel 16.)

```
int    c1, c2;

c1 = 0x45;
c2 = 0x71;
std::cout << "Ergebnis von " << hex << c1 << " & " << c2 << " = " <<
                                (c1 & c2) << dec << '\n';
```

Die Ausgabe dieses Programms lautet:

```
Ergebnis von 45 & 71 = 41
```

Denn:

	c1 = 0x45	binär 01000101
&	c2 = 0x71	binär 01110001
=	0x41	binär 01000001

Das bitweise UND (`&`) ähnelt dem logischen UND (`&&`). Beim logischen UND ist das Ergebnis true (1), wenn beide Operanden true (von null verschieden) sind. Beim bitweisen UND (`&`) ist das entsprechende Bit des Ergebnisses true, wenn die beiden korrespondierenden Bits in den Operanden ebenfalls true (1) sind. Das bitweise UND (`&`) arbeitet also auf jedem Bit einzeln, während das logische UND (`&&`) auf den ganzen Operanden arbeitet.

`&` und `&&` sind aber verschiedene Operatoren, wie Beispiel 11-1 zeigt.

Beispiel 11-1: and/and.cpp

```
#include <iostream>

int main( )
{
    int i1, i2; // zwei zufällige Werte

    i1 = 4;
    i2 = 2;     // Werte setzen

    // Eine nette Möglichkeit, die Bedingung zu formulieren
    if ((i1 != 0) && (i2 != 0))
        std::cout << "Beide sind nicht null #1\n";

    // Eine verkürzte Schreibweise, um das Gleiche zu tun.
    // Korrekter C++-Code, aber lausiger Stil
    if (i1 && i2)
        std::cout << "Beide sind nicht null #2\n";

    // Fehlerhafte Verwendung des Bit-Operators, Fehler
    if (i1 & i2)
        std::cout << "Beide sind nicht null #3\n";
    return (0);
}
```

Frage: Warum gibt Test #3 nicht `Beide sind nicht null #3` aus?

Antwort: Der Operator & ist ein bitweises UND. Das Ergebnis des bitweisen UND ist null:

```
      i1=4    00000100
&     i2=2    00000010
-------------------------
        0     00000000
```

Das Ergebnis des bitweisen UND ist 0, und die Bedingung ist damit nicht erfüllt. Wenn der Programmierer die erste Form

```
if ((i1 != 0) && (i2 != 0))
```

verwendet und aus Versehen & anstelle von && verwendet hätte, wie hier

```
if ((i1 != 0) & (i2 != 0))
```

dann hätte das Programm gleichwohl noch korrekt funktioniert:

```
(i1 != 0)     ist true (Ergebnis = 1)
(i2 != 0)     ist true (Ergebnis = 1)
```

1 bitweise UND 1 ist 1, also ist der Ausdruck wahr.

Kurz, nachdem ich den in diesem Programm dargestellten Bug gefunden hatte, erzählte ich meinem Kollegen: »Jetzt verstehe ich wirklich den Unterschied zwischen UND und UND.« Und er verstand mich. Wie wir Sprache verstehen, hat mich schon immer fasziniert, und die Tatsache, dass ich einen solchen Satz sagen und trotzdem problemlos verstanden werden konnte, hat mich verblüfft.

Sie können den bitweisen UND-Operator verwenden, um herauszufinden, ob eine Zahl gerade oder ungerade ist. In der Basis 2 ist die letzte Ziffer aller geraden Zahlen null und die letzte Ziffer aller ungeraden Zahlen eins. Die folgende Funktion verwendet das bitweise UND, um diese letzte Ziffer herauszupicken. Wenn diese null ist (eine gerade Zahl), dann ist das Ergebnis der Funktion true.

```
inline int gerade(const int wert)
{
    return ((wert & 1) == 0);
}
```

Bitweises ODER (|)

Das inklusive ODER (auch einfach nur als ODER-Operator bezeichnet) vergleicht seine beiden Operanden. Wenn wenigstens eines der beiden Bits 1 ist, dann ist auch das Endergebnis 1. Tabelle 11-4 enthält die Wahrheitstabelle für den ODER-Operator.

Tabelle 11-4: Der bitweise ODER-Operator

Bit1	Bit2	Bit1 | Bit2
0	0	0
0	1	1
1	0	1
1	1	1

Hier sehen Sie ein Beispiel dafür, wie sich ein bitweises ODER auf einem Byte auswirkt:

```
      i1=0x47     01000111
|     i2=0x53     01010011
-------------------------
        0x57      01010111
```

Das bitweise exklusive ODER (^)

Das exklusive ODER (auch XOR oder EOR genannt) hat als Ergebnis 1, wenn genau einer der beiden Operanden 1 ist, aber nicht beide. Die Wahrheitstabelle für das exklusive ODER finden Sie in Tabelle 11-5.

Tabelle 11-5: Der bitweise exklusive ODER-Operator

Bit1	Bit2	Bit1 ^ Bit2
0	0	0
0	1	1
1	0	1
1	1	0

Hier sehen Sie ein Beispiel des exklusivwn ODER auf einem Byte:

```
      i1=0x47     01000111
^     i2=0x53     01010011
-------------------------
        0x14      00010100
```

Der Einerkomplement-Operator (NICHT) (~)

Der NICHT-Operator (auch Invertierungsoperator oder Bit-Schalter genannt) ist ein unärer Operator, der das Inverse seines Operanden zurückgibt, wie Tabelle 11-6 zeigt.

Tabelle 11-6: Der NICHT-Operator

Bit	~Bit
0	1
1	0

Hier sehen Sie ein Beispiel des NICHT-Operators, angewendet auf ein Byte:

c=	0x45	01000101
~c=	0xBA	10111010

Die Verschiebeoperatoren (<<, >>)

Der linke Verschiebeoperator verschiebt die Daten um eine angegebene Anzahl von Bits nach links. Bits, die links herausfallen, verschwinden; rechts wird mit Nullen aufgefüllt. Die Verschiebung nach rechts macht das Gleiche in Gegenrichtung. Ein Beispiel:

	c=0x1C	00011100
c << 1	c=0x38	00111000
c >> 2	c=0x07	00000111

Das Verschieben nach links um eine Position (`x << 1`) ist das Gleiche wie eine Multiplikation mit 2 (`x * 2`). Das Verschieben um zwei Positionen nach links (`x << 2`) ist das Gleiche wie eine Multiplikation mit 4 (`x * 4` oder `x * 2 * 2`). Sie sehen schon das Muster: Das Verschieben um *n* Positionen entspricht der Multiplikation mit 2^n. Warum sollte man verschieben statt multiplizieren? Verschieben ist schneller als das Multiplizieren, daher ist

```
i = j << 3;     // multipliziere j mit 8 (2**3)
```

schneller als:

```
i = j * 8;
```

Oder es wäre zumindest schneller, wenn die Compiler nicht ohnehin schlau genug wären, das »Multiplizieren mit einer Zweierpotenz« in ein »Verschieben« umzuwandeln.

Viele schlaue Programmierer verwenden diesen Trick, um ihre Programme auf Kosten der Lesbarkeit schneller zu machen. Tun Sie das nicht. Der Compiler ist schlau genug, das von selbst zu tun. Wenn Sie eine Verschiebung einbauen, dann gewinnen Sie nichts, verlieren aber Lesbarkeit.

Der linke Verschiebeoperator multipliziert, der rechte dividiert. Daher ist

```
q = i >> 2;
```

das Gleiche wie:

```
q = i / 4;
```

Auch dieser schlaue Trick sollte in modernem Code nicht verwendet werden.

Details zu Verschiebungen nach rechts

Verschiebungen nach rechts sind besonders knifflig. Wenn eine Variable nach rechts verschoben wird, muss C++ den links frei werdenden Platz irgendwie auffüllen. Bei

vorzeichenbehafteten Variablen verwendet C++ den Wert des Vorzeichenbits. Bei vorzeichenlosen Variablen verwendet C++ null. Tabelle 11-7 illustriert einige typische Verschiebungen nach rechts.

Tabelle 11-7: Beispiele zum Verschieben nach rechts

	Vorzeichenbehaftetes Zeichen	Vorzeichenbehaftetes Zeichen	Vorzeichenloses Zeichen
Ausdruck	9 >> 2	–8 >> 2	248 >> 2
Binärer Wert >> 2	0000 1010 >> 2	1111 1000 >> 2	1111 1000 >> 2
Ergebnis	??00 0010	??11 1110 >> 2	??11 1110 >> 2
Auffüllung	Vorzeichenbit (0)	Vorzeichenbit (1)	null
Endergebnis (binär)	0000 0010	1111 1110	0011 1110
Endergebnis (short int)	2	–2	62

Bits setzen, löschen und abfragen

Ein Zeichen enthält acht Bits.[1] Jedes von ihnen kann als separater Schalter (Flag) betrachtet werden. Mit Bitoperationen kann man acht einzelne Bitinformationen in ein einziges Byte packen. Nehmen Sie beispielsweise an, dass Sie ein maschinennahes Kommunikationsprogramm schreiben. Sie speichern die Zeichen zur späteren Verwendung in einem 8 kByte großen Buffer. Zu jedem Zeichen speichern Sie auch eine Reihe von Status-Flags. Diese sind in Tabelle 11-8 aufgeführt.

Tabelle 11-8: Werte für den Kommunikationsstatus

Name	Beschreibung
ERROR	Wahr, falls irgendein Fehler vorliegt.
FRAMING_ERROR	Ein Rahmenfehler ist bei diesem Zeichen aufgetreten.
PARITY_ERROR	Das Zeichen hatte die falsche Parität.
CARRIER_LOST	Das Carrier-Signal ging verloren.
CHANNEL_DOWN	Stromausfall auf dem Kommunikationsgerät.

Sie könnten jedes Flag in einer eigenen Zeichenvariablen speichern. Damit würden Sie aber pro gespeichertem Zeichen fünf weitere Bytes für den Zustand benötigen. Da kommt bei größeren Buffern einiges zusammen. Indem Sie stattdessen jedem Status-Flag ein Bit in einem Acht-Bit-Status-Zeichen zuweisen, reduzieren Sie den Speicherbedarf auf 1/5 des ursprünglichen Bedarfs.

Sie können die Flags den Bitnummern gemäß Tabelle 11-9 zuweisen.

1 Dies gilt für alle Rechner, die ich heutzutage benutze, aber der C++-Standard schreibt nicht vor, wie viele Bits sich in einem Zeichen befinden müssen.

Tabelle 11-9: Bit-Zuweisungen

Bit	Name
0	ERROR
1	FRAMING_ERROR
2	PARITY_ERROR
3	CARRIER_LOST
4	CHANNEL_DOWN

Bits werden konventionsgemäß in der Reihenfolge 76543210 nummeriert. Die Konstanten für die einzelnen Bits sind in Tabelle 11-10 definiert.

Tabelle 11-10: Bitwerte

Bit	Binärer Wert	Hexkonstante
7	10000000	0x80
6	01000000	0x40
5	00100000	0x20
4	00010000	0x10
3	00001000	0x08
2	00000100	0x04
1	00000010	0x02
0	00000001	0x01

Das Folgende ist eine Möglichkeit, die Konstanten für die Bits zu definieren, die den Kommunikationsstatus bilden:

```
// Wahr, wenn irgendein Fehler aufgetreten ist
const int ERROR = 0x01;

// Ein Rahmenfehler ist bei diesem Zeichen aufgetreten
const int FRAMING_ERROR = 0x02;

// Das Zeichen hatte die falsche Parität
const int PARITY_ERROR = 0x04;

// Das Carrier-Signal ging verloren
const int CARRIER_LOST = 0x08;

// Stromausfall auf dem Kommunikationsgerät
const int CHANNEL_DOWN = 0x10;
```

Diese Methode, Bits zu definieren, ist etwas verwirrend. Können Sie (ohne in der Tabelle nachzuschauen) sagen, welche Bitnummer von der Konstante 0x10 repräsentiert wird?

Tabelle 11-11 zeigt, wie man den linken Verschiebeoperator zur Definition der Bits verwenden kann.

Tabelle 11-11: Der linke Verschiebeoperator und die Bitdefinition

C++-Repräsentation	Basis 2-Äquivalent	Ergebnis (Basis 2)	Bitnummer
`1 << 0`	00000001 << 0	00000001	Bit 0
`1 << 1`	00000001 << 1	00000010	Bit 1
`1 << 2`	00000001 << 2	00000100	Bit 2
`1 << 3`	00000001 << 3	00001000	Bit 3
`1 << 4`	00000001 << 4	00010000	Bit 4
`1 << 5`	00000001 << 5	00100000	Bit 5
`1 << 6`	00000001 << 6	01000000	Bit 6
`1 << 7`	00000001 << 7	10000000	Bit 7

Es ist schwierig zu sagen, welches Bit von 0x10 repräsentiert wird, aber leicht zu verstehen, welches Bit mit `1 << 4` gemeint ist.

Eine weitere Möglichkeit, die Konstanten zum Testen der Bits für den Kommunikationsstatus zu definieren, wäre:

```
// Wahr, wenn irgendein Fehler aufgetreten ist
const int ERROR =           (1 << 0);

// Ein Rahmenfehler ist bei diesem Zeichen aufgetreten
const int FRAMING_ERROR =   (1 << 1);

// Das Zeichen hatte die falsche Parität
const int PARITY_ERROR =    (1 << 2);

// Das Carrier-Signal ging verloren
const int CARRIER_LOST =    (1 << 3);

// Stromausfall auf dem Kommunikationsgerät
const int CHANNEL_DOWN =    (1 << 4);
```

Jetzt haben Sie die Bits definiert und können sie manipulieren. Zum Setzen eines Bits verwendet man den Operator |. Ein Beispiel:

```
char    flags = 0;  // Alle Flags beginnen mit 0

    flags |= CHANNEL_DOWN; // Kanal ging gerade verloren
```

Um ein Bit abzufragen, verwenden Sie den Operator &, um die Bits »auszumaskieren«:

```
    if ((flags & ERROR) != 0)
        std::cerr << "Fehler-Flag ist gesetzt\n";
    else
        std::cerr << "Kein Fehler gemeldet\n";
```

Das Löschen ist ein klein wenig schwieriger. Angenommen, Sie wollen das Bit `PARITY_ERROR` löschen. Binär ist dieses Bit 00000100. Sie brauchen eine Maske, bei der alle

Bits außer diesem gesetzt sind (11111011). Dies geschieht mit dem NICHT-Operator (~). Die Maske wird dann mit der Zahl, in der das Bit gelöscht werden soll, UND-verknüpft.

PARITY_ERROR	00000100
~PARITY_ERROR	11111011
flags	00000101
flags & ~PARITY_ERROR	00000001

In C++ sieht das so aus:

```
flags &= ~PARITY_ERROR; // Wer interessiert sich für Parität?
```

Frage 11-1: *Im folgenden Programm funktioniert das Flag* HIGH_SPEED*, das Flag* DIRECT_CONNECT *aber nicht. Warum?*

```
#include <iostream>

const int HIGH_SPEED = (1<<7);    /* Modem läuft schnell */

const int DIRECT_CONNECT = (1<<8);  // hart verdrahtete Verbindung

char flags = 0;                     // bei null anfangen

int main( )
{
    flags |= HIGH_SPEED;            // Modem läuft schnell
    flags |= DIRECT_CONNECT;        // weil wir direkt verdrahtet sind

    if ((flags & HIGH_SPEED) != 0)
        std::cout <<"Höchstgeschwindigkeit\n";

    if ((flags & DIRECT_CONNECT) != 0)
        std::cout <<"Direkte Verbindung\n";
    return (0);
}
```

Bitmap-Grafiken

In schwarzweißen Bitmap-Grafiken wird jedes Pixel im Bild durch ein einziges Bit im Speicher repräsentiert. Beispielsweise zeigt Abbildung 11-1 ein 14-mal-14-Bitmap-Bild, einmal so, wie es auf dem Bildschirm erscheint, und einmal vergrößert, so dass Sie die Bits sehen können.

Angenommen, wir haben ein kleines grafisches Ausgabegerät, eine einfarbige Anzeige mit 16 mal 16 Pixeln. Wir wollen das Bit an der Position (4, 7) setzen. Die Bitmap für dieses Gerät ist in Abbildung 11-2 als Array von Bits zu sehen.

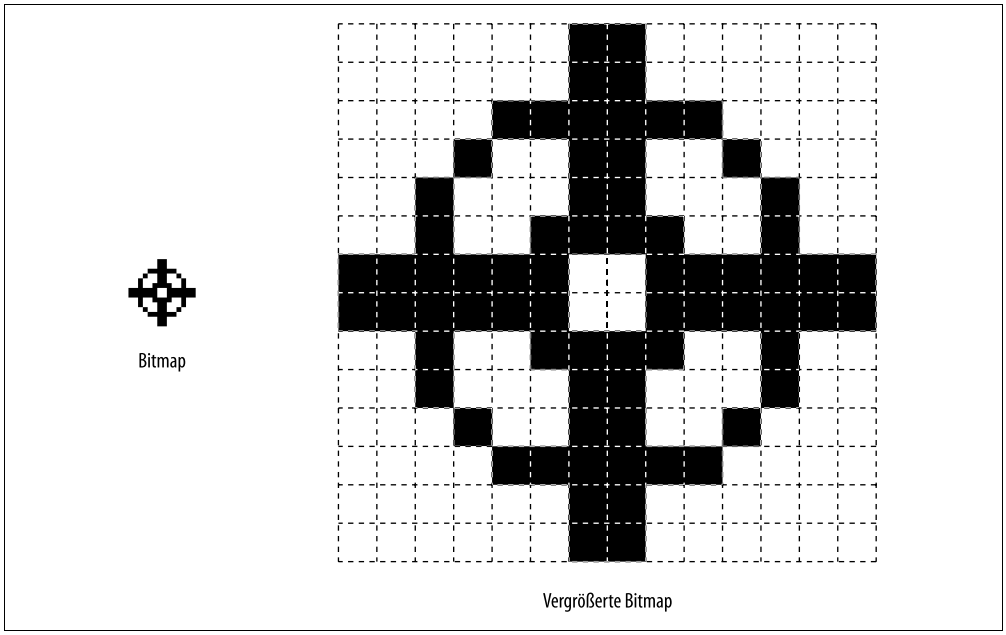

Abbildung 11-1: Bitmap, tatsächliche Größe und vergrößert

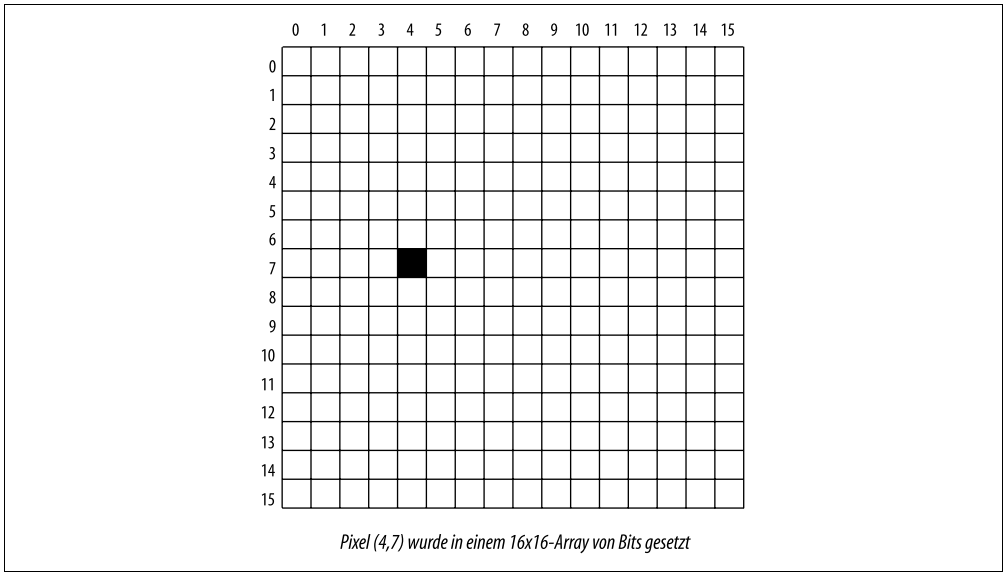

Abbildung 11-2: Array von Bits

Aber wir haben hier ein Programm. Es gibt in C++ keinen Datentyp für Arrays von Bits. Am nähesten kommt noch ein Array von Bytes. Unser 16-mal-16-Bit-Array wird nun zu einem 2-mal-16-Byte-Array, wie es in Abbildung 11-3 zu sehen ist.

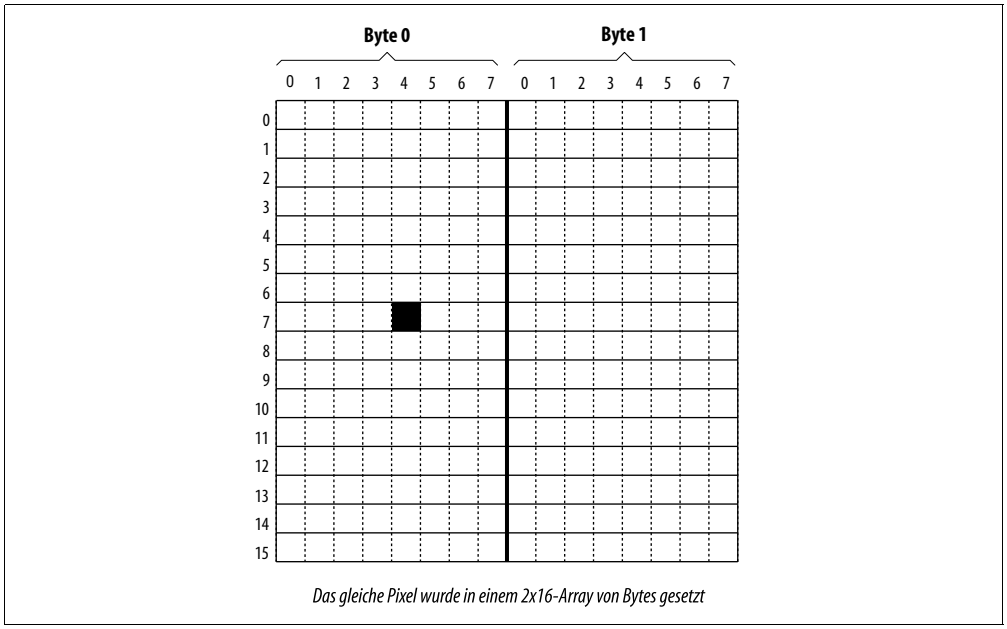

Abbildung 11-3: Array von Bytes

Um das Pixel an Position (4, 7) zu setzen, müssen wir das vierte Bit von Byte (0, 7) setzen. Um dieses Bit zu setzen, würden wir die Anweisung `bit_array[0][7] |= (0x80 >> (4));` verwenden (die Konstante `0x80` ist das am weitesten links stehende Bit).

Wir verwenden in diesem Fall die Notation `(0x80 >> (4))`, um das vierte Bit von links (eine Pixelposition) zu repräsentieren. Bisher haben wir `(1 << 4)` verwendet, um über das vierte Bit von rechts (eine Bitnummer) zu sprechen.

Wir können den Vorgang des Setzens von Pixeln mit einer Funktion verallgemeinern, die das Bit (Pixel) an der Position (*x*, *y*) einschaltet. Wir müssen dazu zwei Werte berechnen: die Koordinate des Bytes und die Nummer des Bits in diesem Byte.

Unsere Bitadresse ist (*x*, *y*). Bytes sind Gruppen von acht Bits, also ist unsere Byteadresse (*x*/8, *y*).

Das Bit im Byte ist nicht so einfach. Wir wollen eine Maske erzeugen, die aus dem Bit besteht, das wir setzen wollen. Für das am weitesten links stehende Bit sollte das 1000 0000 oder 0x80 sein. Das ist der Fall, wenn `(x%8) == 0`. Das nächste Bit ist 0100 0000 oder `(0x80 >> 1)` und kommt vor, wenn `(x%8) == 1`. Um unsere Bitmaske zu generieren, verwenden wir daher den Ausdruck `(0x80 >> (x%8))`.

Jetzt haben wir die Byteposition und die Bitmaske und müssen nur noch das Bit setzen. Die folgende Funktion setzt ein gegebenes Bit in einem Bitmap-Grafik-Array namens *graphics*:

```
void inline set_bit(const int x,const int y)
```

```
    {
        assert((x >= 0) && (x < X_SIZE));
        assert((y >= 0 )) && (y < y_SIZE);
        graphics[x/8][y] | = (0x80) >> (x%8));
    }
```

Beispiel 11-2 zieht eine diagonale Linie über das Grafik-Array und gibt dann das Array auf der Konsole aus.

Beispiel 11-2: graph/graph.cpp

```
#include <iostream>
#include <assert.h>

const int X_SIZE = 40; // Größe des Arrays in X-Richtung
const int Y_SIZE = 60; // Größe des Arrays in Y-Richtung
/*
 * Wir verwenden X_SIZE/8, weil wir 8 Bits per Byte packen
 */
char graphics[X_SIZE / 8][Y_SIZE];   // die Grafikdaten

/*******************************************************
 * set_bit -- ein Bit im Grafik-Array setzen.          *
 *                                                     *
 * Parameter                                           *
 *      x,y -- Position des Bits.                      *
 *******************************************************/
inline void set_bit(const int x,const int y)
{
    assert((x >= 0) && (x < X_SIZE));
    assert((y >= 0) && (y < Y_SIZE));
    graphics[(x)/8][y] |= static_cast<char>(0x80 >>((x)%8));
}

int main( )
{
    int   loc;              // aktuelle zu setzende Position
    void  print_graphics( ); // die Daten ausgeben

    for (loc = 0; loc < X_SIZE; ++loc)
        set_bit(loc, loc);

    print_graphics( );
    return (0);
}
/*******************************************************
 * print_graphics -- das Grafik-Bit-Array als Folge von *
 *                   X und . ausgeben.                 *
 *******************************************************/
void print_graphics( )
{
    int x;     // aktuelles x-Byte
    int y;     // aktuelle y-Position
    int bit;   // das Bit, das wir im aktuellen Byte abfragen
```

Beispiel 11-2: graph/graph.cpp (Fortsetzung)

```
    for (y = 0; y < Y_SIZE; ++y) {

        // Schleife über alle Bytes im Array
        for (x = 0; x < X_SIZE / 8; ++x) {

            // Alle Bits verarbeiten
            for (bit = 0x80; bit > 0; bit = (bit >> 1)) {
                assert((x >= 0) && (x < (X_SIZE/8)));
                assert((y >= 0) && (y < Y_SIZE));

                if ((graphics[x][y] & bit) != 0)
                    std::cout << 'X';
                else
                    std::cout << '.';
            }
        }
        std::cout << '\n';
    }
}
```

Das Programm definiert ein Bitmap-Grafik-Array:

```
    char graphics[X_SIZE / 8][Y_SIZE];   // die Grafikdaten
```

Wir verwenden die Konstante `X_SIZE/8`, weil wir horizontal `X_SIZE` Bits haben, was dann `X_SIZE/8` Bytes ergibt.

Die **for**-Schleife

```
    for (loc = 0; loc < X_SIZE; ++loc)
            set_bit(loc, loc);
```

zeichnet eine diagonale Linie über das Grafik-Array.

Weil wir kein Bitmap-Grafik-Gerät haben, simulieren wir es mit der Funktion `print_graphics`.

Die folgende Schleife gibt alle Zeilen aus:

```
    for (y = 0; y < Y_SIZE; ++y) {
        ....
```

Diese Schleife durchläuft alle Bytes in einer Zeile:

```
        for (x = 0; x < X_SIZE / 8; ++x) {
            ...
```

Die acht Bits in jedem Byte werden von folgender Schleife durchlaufen:

```
            for (bit = 0x80; bit > 0; bit = (bit >> 1))
```

Hier wird ein ungewöhnlicher Schleifenzähler verwendet. In dieser Schleife beginnt die Variable `bit` mit dem Bit 7 (dem am weitesten links stehenden Bit). In jedem Schleifendurchlauf verschiebt `bit = (bit >> 1)` das Bit um eine Position nach rechts. Wenn keine Bits mehr übrig sind, endet die Schleife.

Der Schleifenzähler durchläuft die folgenden Werte:

Binär	Hex
1000 0000	0x80
0100 0000	0x40
0010 0000	0x20
0001 0000	0x10
0000 1000	0x08
0000 0100	0x04
0000 0010	0x02
0000 0001	0x01

Schließlich befindet sich der folgende Code im Herzen aller dieser Schleifen:

```
if ((graphics[x][y] & bit) != 0)
    std::cout <<"X";
else
    std::cout << ".";
```

Hier wird ein einzelnes Bit abgefragt und »X« ausgegeben, wenn das Bit gesetzt ist, beziehungsweise ».«, wenn es nicht gesetzt ist.

Frage 11-2: *In Beispiel 11-3 funktioniert die erste Schleife, die zweite aber nicht. Warum?*

Beispiel 11-3: loop/loop.cpp

```
#include <iostream>

int main( )
{
    short int i;

    // Funktioniert
    for (i = 0x80; i != 0; i = (i >> 1)) {
        std::cout << "i ist " << std::hex << i << std::dec << '\n';
    }

    signed char ch;

    // Funktioniert nicht
    for (ch = 0x80; ch != 0; ch = (ch >> 1)) {
        std::cout << "ch ist " << std::hex <<
            static_cast<int>(ch) << std::dec << '\n';
    }
```

Beispiel 11-3: loop/loop.cpp (Fortsetzung)

```
    return (0);
}
```

Programmieraufgaben

Übung 11-1: Schreiben Sie einen Satz von Inline-Funktionen namens `clear_bit` und `test_bit`, die zur in Beispiel 11-2 definierten Operation `set_bit` passen. Schreiben Sie ein Hauptprogramm, das diese drei Funktionen testet.

Übung 11-2: Schreiben Sie ein Programm, das ein 10-mal-10-Bitmap-Quadrat zeichnet.

Übung 11-3: Ändern Sie Beispiel 11-1 dahingehend, dass eine weiße Linie auf schwarzem Hintergrund gezeichnet wird.

Übung 11-4: Schreiben Sie ein Programm, das die Anzahl der gesetzten Bits in einem Integer-Wert zählt. Beispielsweise sind in der Zahl 5 (dezimal), die in binärer Notation 0000000000000101 ist, zwei Bits gesetzt.

Übung 11-5: Schreiben Sie ein Programm, das einen 32-Bit-Integer-Wert (**long int**) erwartet und diesen in acht 4-Bit-Werte aufteilt (beachten Sie das Vorzeichenbit).

Übung 11-6: Schreiben Sie ein Programm, das alle Bits in einer Zahl an das linke Ende verschiebt. Beispielsweise würde binär 01010110 zu binär 11110000 werden.

Antworten zu den Fragen in diesem Kapitel

Antwort 11-1: `DIRECT_CONNECT` wird durch den Ausdruck `(1 << 8)` als Bit Nummer 8 definiert; die acht Bits in einer Zeichenvariablen sind aber als 76543210 nummeriert. Es gibt kein Bit mit der Nummer acht. Man kann dieses Problem lösen, indem man `flags` zu einem **short int** mit 16 Bits macht.

Antwort 11-2: Das Problem besteht darin, dass `ch` ein Zeichen (8 Bits) ist. Der Wert 0x80, repräsentiert in 8 Bits, ist 1000 0000. Das erste Bit, das Vorzeichenbit, ist gesetzt. Wenn diese Variable nach rechts verschoben wird, dann wird das Vorzeichenbit zum Füllen verwendet, so dass 1000 0000 >> 1 gleich 1100 0000 ist.

Die Variable `i` funktioniert, obwohl sie vorzeichenbehaftet ist, weil sie 16 Bits breit ist. 0x80 in 16 Bits ist 0000 0000 1000 0000. Beachten Sie, dass sich das Bit, das wir gesetzt haben, weit entfernt vom Vorzeichenbit befindet.

Die Lösung zu diesem Problem besteht darin, `ch` als **unsigned**-Variable zu verwenden.