CHAPTER 5

OPERATORS

C OPERATORS

OPERATOR	DESCRIPTION
x[i], **f(x)**	array subscripting and function call
., **->**	direct and indirect structure field selection
++, **--**	postfix increment/decrement
++, **--**	prefix increment/decrement
sizeof, **(***type***)**	size of a variable or type (in bytes), cast to *type*
-, **!**, **~**	unary minus, logical and bitwise NOT
&, *****	address of and dereferencing
*****, **/**, **%**	multiply, divide, modulus
+, **-**	addition, subtraction
>>, **<<**	right, left shift
<, **>**, **<=**, **>=**	test for inequality
==, **!=**	test for equality, inequality
&	bitwise AND
^	bitwise XOR (exclusive OR)
|	bitwise OR
&&, **||**	logical AND and logical OR
? :	conditional operator
=	assignment
+=, **-=**, ***=**, **/=**, **%=**	add to, subtract from, multiply to, divide by, assign remainder
<<=, **>>=**, **^=**, **&=**, **|=**	shift right and left, assign bitwise XOR, AND, OR
,	sequential expression evaluation

The Joy Of C, ©1993 by John Wiley & Sons, Inc.

brkwd.c

(Figure 5.1 – Page 93)

```
/*
 * Break input into words (groups of letters).
 */
#include <stdio.h>
#include <ctype.h>

#define YES  1
#define NO   0

int main()
{
  int c;                        /* next input character */
  int word;                     /* flag: are we dealing with a word? */

  word = NO;
  while ((c = getchar()) != EOF)
    if (isalpha(c))
    {                           /* it's a letter */
      putchar(c);               /* write it and note we're in word */
      word = YES;
    }
    else                        /* it's not a letter */
      if (word)                 /* write a newline if we were in word */
      {
        putchar('\n');
        word = NO;
      }
  return 0;
}
```

avg.c

(Figure 5.2 – Page 95)

```
/*
 * Compute average of its input values.
 */
#include <stdio.h>

int main()
{
  int    next;                       /* next input value */
  long   sum;                        /* running total */
  int    n;                          /* number of input values */
  int    result;                     /* did we read another value? */
  double avg;                        /* average of input values */

  sum = 0;
  n = 0;
  while ((result = scanf("%i", &next)) == 1)
  {
    sum = sum + next;
    n = n + 1;
  }
  if (result != EOF)
    printf("Warning: bad input after reading %i values\n", n);
  if (n = 0)                         /* MISTAKE! */
    avg = 0.0;
  else
    avg = (double) sum / n;
  printf("Average of %i values is %f.\n", n, avg);

  return 0;
}
```

square.c

(Not In Text)

```
/*
 * Determine whether a point is inside a square.
 */
#include <stdio.h>

int inside_square(double x, double y,   /* point */
                  double sx, double sy, /* center of the square */
                  double side)          /* length of a square's side */
{
  int between(double low, double high, double val);
  double half_side;                     /* distance from center to side */

  half_side = side / 2;
  return between(sx - half_side, sx + half_side, x)
           &&
         between(sy - half_side, sy + half_side, y);
}

int between(double low, double high, double val)
{
  return val >= low && val <= high;
}
```

count.c

(Not In Text)

```
/*
 * Count blanks and tabs in the input.
 */
#include <stdio.h>

int main()
{
  long charcnt;            /* character counter */
  long spacecnt;           /* blank and tab counter */
  int  c;                  /* next character */

  spacecnt = 0;
  charcnt = 0;
  while ((c = getchar()) != EOF)
  {
    charcnt = charcnt + 1;
    if (c == ' ' || c == '\t')
      spacecnt = spacecnt + 1;
  }
  printf("%li blanks or tabs out of %li chars.\n", spacecnt, charcnt);
  return 0;
}
```

BEHAVIOR OF LOGICAL OPERATORS

OPERANDS		RESULT	
OP1	OP2	&&	||
nonzero	nonzero	1	1
nonzero	zero	0	1
zero	nonzero	0	1
zero	zero	0	0

yesorno.c

(Figure 5.3 – Page 96)

```
/*
 * Print a prompt and then get a yes or no answer using "yesorno".
 */
#include <stdio.h>
#include <ctype.h>

#define YES 1
#define NO  0

int main()
{
  int yesorno(void);

  printf("Enter a YES or NO answer: ");
  if (yesorno() == YES)
    printf("That was a YES!\n");
  else
    printf("That was a NO!\n");
  return 0;
}

int yesorno(void)
{
  int c;                             /* holds input character */
  int answer;                        /* holds YES or NO answer */

  if ((c = getchar()) == 'y' || c == 'Y')
    answer = YES;                    /* YES for 'Y' or 'y' */
  else
    answer = NO;                     /* NO for anything else */
  while (c != EOF && c != '\n')    /* skip other characters on line */
    c = getchar();
  return answer;
}
```

yesorno2.c

(Figure 5.4 – Page 97)

```
/*
 * Print a prompt and then get a yes or no answer using "yesorno".
 */
#include <stdio.h>
#include <ctype.h>

int main()
{
  int yesorno(void);

  printf("Enter a YES or NO answer: ");
  if (yesorno())
    printf("That was a YES!\n");
  else
    printf("That was a NO!\n");
  return 0;
}

int yesorno(void)
{
  int c;                            /* holds input character */
  int answer;                       /* holds YES or NO answer */

  answer = (c = getchar()) == 'y' || c == 'Y';
  while (c != EOF && c != '\n')     /* skip other characters on line */
    c = getchar();
  return answer;                    /* 1 for 'Y' or 'y', otherwise 0 */
}
```

Sample I/O – brkwd2

```
w8 4 me 2go with you.
_ _ _ _ _
```

```
w8
me
go
with
you
```

brkwd2.c

(Figure 5.5 – Page 98)

```
/*
 * Break input into words (letter followed by letters or digits).
 */
#include <stdio.h>
#include <ctype.h>

#define YES  1
#define NO   0

int main()
{
  int c;                          /* next input character */
  int word;                       /* flag: are we dealing with a word? */

  word = NO;
  while ((c = getchar()) != EOF)
    if (isalpha(c) || (word && isdigit(c)))
    {                             /* it's a letter */
      putchar(c);                 /* write it and note we're in word */
      word = YES;
    }
    else                          /* it's not a letter */
      if (word)                   /* write a newline if we were in word */
      {
        putchar('\n');
        word = NO;
      }
  return 0;
}
```

shrink.c

(Figure 5.6 – Page 100)

```
/*
 * Removes leading and trailing whitespace from its input and turns runs
 * of whitespace into a single blank.  Assumes lines end with a newline.
 */
#include <stdio.h>
#include <ctype.h>

int main()
{
  int c;                                /* next input character */

  while ((c = getchar()) != EOF)
  {            /* each pass through loop takes care of one input line */
    while (isspace(c) && c != '\n')
      c = getchar();                    /* skip leading spaces */
    while (c != '\n')
    {                                   /* process any words */
      while (!isspace(c))
      {                                 /* dump out word */
        putchar(c);
        c = getchar();
      }
      while (isspace(c) && c != '\n')   /* skip trailing spaces */
        c = getchar();
      if (c != '\n')                    /* not trailing, write space */
        putchar(' ');
    }
    putchar('\n');                      /* trailing, write newline */
  }
  return 0;
}
```

EFFECT OF BIT-SHIFT OPERATORS

`x`	`x << 1`	`x >> 1`
0...0 0110	0...0 1100	0...0 0011
6	12	3

THE BITWISE LOGICAL OPERATORS

Operands		Results		
Op1	Op2	&	|	^
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

EXAMPLE OF BITWISE LOGICAL OPERATORS

EXPRESSION	BINARY VALUE			
A:	1001	0110	0001	0001
B:	0001	1010	1101	1001
A & B:	0001	0010	0001	0001
A | B:	1001	1110	1101	1001
A ^ B:	1000	1100	1100	1000
~A:	0110	1001	1110	1110
~B:	1110	0101	0010	0110

Sample I/O – prbin

```
Enter value:  15487
In binary: 0011110001111111
15487 >> 1 = 7743: 0001111000111111
15487 << 1 = 30974: 0111100011111110
Bit 7 was off, now on = 15615: 0011110011111111
```

prbin.c

(Figure 5.7 – Page 104)

```
/*
 * A pair of useful bit-manipulating functions.
 *   intbits - returns number of bits in an unsigned int.
 *   print_binary - prints an unsigned int in binary.
 */
#include <stdio.h>

int int_bits(void)
{
  int x;                                        /* word being shifted */
  int i;                                        /* count of shifts */

  i = 0;
  for (x = 1; x != 0; x = x << 1)
    i = i + 1;
  return i;
}

void print_binary(unsigned int x, int nbits)
{
  int i;                                        /* index of desired bits */

  for (i = nbits - 1; i >= 0; i = i - 1)
    printf("%i", (x >> i) & 01);
  putchar('\n');
}
```

useprbin.c

(Figure 5.8 – Page 105)

```
/*
 * Program illustrating some basic bit-manipulating operations.
 */
#include <stdio.h>

#define OURBIT  0x80                     /* bit 7: (...00 1000 0000) */

int main()
{
  int      int_bits(void);
  void     print_binary(unsigned value, int intbits);
  int      nbits;                        /* bits in an int */
  unsigned value;                        /* input value */
  unsigned x;                            /* modified input value */

  nbits = int_bits();                    /* get # of bits in an int */
  printf("Enter value: ");               /* get value to display */
  scanf("%u", &value);                   /* assume no errors */

  printf("In binary: ");
  print_binary(value, nbits);            /* display in binary */
  x = value >> 1;
  printf("%u >> 1 = %u: ", value, x);    /* show right shifted */
  print_binary(x, nbits);
  x = value << 1;                        /* show left shifted */
  printf("%u << 1 = %u: ", value, x);
  print_binary(x, nbits);

  if ((value & OURBIT) != 0)             /* show with bit 7 off and on */
    printf("Bit 7 was on, now off: ", x = value & ~ OURBIT);
  else
    printf("Bit 7 was off, now on = %u: ", x = value | OURBIT);
  print_binary(x, nbits);
  return 0;
}
```

bits.c

(Figure 5.9 – Page 106)

```
/*
 * Functions to get and set bit values.  All assume suitable parameters.
 *   getbit:  get value of bit n in word.
 *   setbit:  return value with bit n set to v (0 or 1).
 *   getbits: get value of k bits at position n in word.
 *   setbits: return value with k bits starting at bit n assigned v.
 */

unsigned getbit(unsigned word, int n)
{
  return (word >> n) & 01;
}

unsigned setbit(unsigned word, int n, unsigned v)
{
  if (v != 0)
    return word | (01 << n);    /* turn on the bit */
  else
    return word & ~(01 << n);   /* turn off the bit */
}

unsigned getbits(unsigned word, int n, int k)
{
   return (word & (~(~0 << k) << n)) >> n;
}

unsigned setbits(unsigned word, int n, int k, unsigned v)
{
   return (word & ~(~(~0 << k) << n)) | (v << n);
}
```

HOW GETBITS WORKS

Assume call is **`getbits(word,3,2)`** with **`word`** equal to 181

Build mask (**`n`**=3, **`k`**=2)

all ones (˜0):	`1111111111111111`
`k` zeros (**`<< k`**):	`1111111111111100`
invert:	`0000000000000011`
shift (**`<< n`**):	`0000000000011000`

AND with **`word`** and shift

`word`:	`0000000010110101`
after AND:	`0000000000010000`
after shift (**`>> n`**):	`0000000000000010`

HOW SETBITS WORKS

Assume call is **`setbits(word,2,5,16)`** with **`word`** equal to 181

Build mask (**`n`**=2, **`k`**=5)

earlier mask:	**000000000**11111**00**
inverted:	**111111111**00000**11**

AND with **`word`**, OR with shifted value

`word`:	**00000000**10110**101**
after AND:	**000000001**00000**01**
shifted value:	**000000000**10000**00**
after OR:	**000000001**10000**01**

EXAMPLE BIT MANIPULATIONS

CALL	RETURN VALUE	BIT PATTERN
`getbit(word, 1)`	0	`0...0 0000 0000`
`getbit(word, 2)`	1	`0...0 0000 0001`
`setbit(word, 3, 1)`	189	`0...0 1011 1101`
`setbit(word, 4, 0)`	165	`0...0 1010 0101`
`getbits(word, 3, 2)`	2	`0...0 0000 0010`
`getbits(word, 0, 4)`	5	`0...0 0000 0101`
`setbits(word, 2, 5, 16)`	193	`0...0 1100 0001`
`setbits(word, 0, 4, 7)`	183	`0...0 1011 0111`

Assume **`word`** is 181 or **`0...0 1011 0101`** in binary.

usebits.c

(Figure 5.12 – Page 109)

```
/*
 * Pack employee information into a single word (no error checking).
 */
#include <stdio.h>

#define SINGLE    0          /* codes for marital status */
#define MARRIED   1
#define SEPARATED 2
#define DIVORCED  3
#define MALE      1          /* codes for sex */
#define FEMALE    0
#define MSBIT     14         /* starting bit for marital status */
#define MSBITS    2          /* number of marital status bits */
#define AGEBIT    7          /* starting bit for age */
#define AGEBITS   7          /* number of bits for age */
#define SEXBIT    6          /* bit number for sex */
#define YRSBIT    0          /* starting bit for employed years */
#define YRSBITS   6          /* number of bits for years */
```

usebits.c

(Figure 5.12 – Page 109)

```
int main()
{
  unsigned getbit(unsigned word, int n);
  unsigned setbit(unsigned word, int n, unsigned v);
  unsigned getbits(unsigned word, int n, int k);
  unsigned setbits(unsigned word, int n, int k, unsigned v);
  unsigned mstat, sex, age, years;
  unsigned short info;          /* holds info on one person */

  info = 0;                     /* all bits off initially */
  printf("male=%i, female=%i? ", MALE, FEMALE);
  scanf("%u", &sex);
  info = setbit(info, SEXBIT, sex);
  printf("age? ");
  scanf("%u", &age);
  info = setbits(info, AGEBIT, AGEBITS, age);
  printf("single=%u, married=%u, separated=%u, divorced=%u? ",
         SINGLE, MARRIED, SEPARATED, DIVORCED);
  scanf("%u", &mstat);
  info = setbits(info, MSBIT, MSBITS, mstat);
  printf("years employed? ");
  scanf("%u", &years);
  info = setbits(info, YRSBIT, YRSBITS, years);
  printf("Sex: %u\n", getbit(info, SEXBIT));
  printf("Age: %u\n", getbits(info, AGEBIT, AGEBITS));
  printf("Marital status: %u\n", getbits(info, MSBIT, MSBITS));
  printf("Years employed: %u\n", getbits(info, YRSBIT, YRSBITS));
  return 0;
}
```

COMPRESSING DATA INTO A WORD

15 ··· 14	13 ··· 7	6	5 ··· 0
STATUS	AGE	SEX	YEARS

The Joy Of C, ©1993 by John Wiley & Sons, Inc.

XOR-BASED ENCRYPTION

`DATA (ASCII 'P')`	0101	0000	
`KEY`	0001	0010	
`DATA xor KEY`	0100	0010	(encrypted data)

`DATA xor KEY`	0100	0010	(encrypted data)
`KEY`	0001	0010	
`DATA`	0101	0000	

encrypt.c
(Figure 5.13 – Page 110)

```
/*
 * Encrypt the standard input using a built-in encryption key (0xABCD).
 */
#include <stdio.h>
#include <limits.h>

#define MAXKEY     4L     /* length of the key (4 bytes) */
#define HIGHBIT  0x80     /* mask to access high bit only */

int main()
{
  unsigned long key;      /* the encryption key */
  unsigned char keybyte;  /* the next byte in the key */
  int           c;        /* next character */
  unsigned long j;        /* character count */

  key = 0xABCD;           /* hex key: byte 1 = 10, byte 2 = 11, etc. */
  j = 0;
  while ((c = getchar()) != EOF)
  {
    keybyte = (unsigned char) (key >> ((int) (j % MAXKEY) * CHAR_BIT));
    putchar(c ^ (keybyte | HIGHBIT));
    j = j + 1;
  }
  return 0;
}
```

power.c

(Figure 5.14 – Page 112)

```
/*
 * Compute powers, more concise version.
 */
double power(double x, int exp)
{
  double p;

  p = 1.0;
  if (exp > 0)                              /* positive exponent */
    while (exp > 0)
    {
      p *= x;
      exp -= 1;
    }
  else                                      /* negative exponent */
    while (exp < 0)
    {
      p /= x;
      exp += 1;
    }
  return p;
}
```

power2.c

(Figure 5.15 – Page 113)

```
/*
 * Most concise version of our power computing function.
 */
double power(double x, int exp)
{
  double p;

  p = 1.0;
  if (exp > 0)                                  /* positive exponent */
    while (exp > 0)
    {
      p *= x;
      exp--;
    }
  else                                          /* negative exponent */
    while (exp < 0)
    {
      p /= x;
      exp++;
    }
  return p;
}
```

power3.c

(Not In Text)

```
/*
 * An extremely concise version of our power computing function.
 */
double power(double x, int exp)
{
  double p;

  p = 1.0;
  if (exp > 0)                                  /* positive exponent */
    while (exp-- > 0)
      p *= x;
  else                                          /* negative exponent */
    while (exp++ < 0)
      p /= x;
  return p;
}
```

lineno.c

(Figure 5.16 – Page 114)

```
/*
 * Copy input to output, giving each line a number.
 */
#include <stdio.h>

int main()
{
  int  c;                          /* current and */
  int  lastch;                     /*    previous characters */
  long lineno;                     /* lines printed so far */

  lineno = 0;  lastch = '\n';
  while ((c = getchar()) != EOF)
  {
    if (lastch == '\n')            /* hit end of line */
      printf("%8li ", ++lineno);
    putchar(c);
    lastch = c;
  }
  return 0;
}
```

lineno2.c

(Figure 5.17 – Page 114)

```
/*
 * Copy input to output, giving each line a number.
 */
#include <stdio.h>

int main()
{
  int  c;                        /* current and */
  int  lastch;                   /*    previous characters */
  long lineno;                   /* lines printed so far */

  lineno = 1;  lastch = '\n';
  while ((c = getchar()) != EOF)
  {
    if (lastch == '\n')          /* hit end of line */
      printf("%8li ", lineno++);
    putchar(c);
    lastch = c;
  }
  return 0;
}
```

display.c

(Figure 5.18 – Page 116)

```
/*
 * Copy the input to the output.
 */
#include <stdio.h>

int main()
{
  int c;                                  /* next input character */

  while (c = getchar(), c != EOF)
    putchar(c);
  return 0;
}
```

Sample I/O – usepower

```
Enter x,y:  10 4
10^4 = 10000
Enter x,y:  2.5 8
2.5^8 = 1525.88
Enter x,y:
```

usepower.c

(Figure 5.19 – Page 116)

```
/*
 * Using our own function to compute exponents.
 */
#include <stdio.h>

int main()
{
  double power(double base, int exp);
  double x;                              /* user-supplied base */
  int    y;                              /* user-supplied exponent */

  while (printf("Enter x,y: "), scanf("%lf %i", &x, &y) == 2)
    printf("%g^%i = %g\n", x, y, power(x, y));
  return 0;
}
```

Sample Output – sizeof

```
short:    2
int:      2
long:     4
float:    4
double:   8
```

sizeof.c

(Figure 5.20 – Page 118)

```
/*
 * Print the sizes of various types.
 */
#include <stdio.h>
#include <stddef.h>

int main()
{
  printf("short:   \t%lu\n", (unsigned long) sizeof(short));
  printf("int:     \t%lu\n", (unsigned long) sizeof(int));
  printf("long:    \t%lu\n", (unsigned long) sizeof(long));
  printf("float:   \t%lu\n", (unsigned long) sizeof(float));
  printf("double:  \t%lu\n", (unsigned long) sizeof(double));
  return 0;
}
```

avg2.c

(Figure 5.21 – Page 119)

```
/*
 * Compute average of its input values.
 */
#include <stdio.h>

int main()
{
  int    next;                          /* next input value */
  long   sum;                           /* running total */
  int    n;                             /* number of input values */
  int    result;                        /* did we read another value? */
  double avg;                           /* average of input values */

  sum = n = 0;
  while ((result = scanf("%i", &next)) == 1)
  {
    sum += next;
    n++;
  }
  if (result != EOF)
    printf("Warning: bad input after reading %i values\n", n);
  avg = (n == 0) ? 0.0 : (double) sum / n;
  printf("Average of %i values is %f.\n", n, avg);
  return 0;
}
```

minmax.c
(Not In Text)

```
/*
 * Functions to return their smallest and largest arguments.
 */

int min(int x, int y)
{
  return (x < y) ? x : y;
}

int max(int x, int y)
{
  return (x > y) ? x : y;
}
```

compress.c

(Figure 5.22 – Page 121)

```
/*
 * Simple data-compression program (assumes ASCII characters).
 */
#include <stdio.h>
#include <limits.h>

#define MINSIZE    3          /* smallest group to compress */
#define HIGHBIT 0x80          /* high bit indicates compressed char */

int main()
{
  int c;                      /* last character read */
  int newc;                   /* new character read */
  int cntc;                   /* count of last character */

  c = getchar();
  while (c != EOF)
  {
    cntc = 1;                 /* count occurrences of c */
    while (cntc < UCHAR_MAX && (newc = getchar()) == c)
      cntc++;
    if (cntc >= MINSIZE)
    {                         /* write count (high bit on) */
      putchar(c | HIGHBIT);
      putchar(cntc);
    }
    else                      /* write chars (high bit off) */
      while (cntc-- > 0)
        putchar(c & ~HIGHBIT);
    c = newc;
  }
  return 0;
}
```

uncmprss.c
(Figure 5.23 – Page 122)

```
/*
 * Simple uncompress program.
 */
#include <stdio.h>

#define HIGHBIT   0x80      /* high bit on indicates compressed char */

int main()
{
  int getcount(int c);
  int c;                    /* new character read */
  int cntc;                 /* count of times char should appear */

  while ((c = getchar()) != EOF && (cntc = getcount(c)) != 0)
    while (cntc-- > 0)
      putchar(c & ~HIGHBIT);
  return 0;
}

int getcount(int c)         /* determine # of times to write char */
{
  if (! (c & HIGHBIT))
    return 1;                          /* once - not compressed */
  return ((c = getchar()) == EOF) ? 0  /* zero - EOF followed char */
                                  : c; /* next char is count */
}
```

REMINDERS

- Don’t forget to parenthesize assignments when using the relational operators to compare their results.

- Watch out for accidentally using = to test for equality.

- Unlike the relational operators, `&&` and `||` impose an order of evaluation on their operands.

- Whenever possible, avoid right shifts on signed values.

- Parenthesize all uses of the bitwise operators in larger expressions.

- The comma separating parameters in function calls is not a comma operator and does not guarantee left-to-right evaluation.

- Avoid writing nested conditional operators.

CHAPTER SUMMARY

- C provides relational operators for testing whether one value is equal (`==`), not equal (`!=`), less than (`<`), greater than (`>`), less than or equal (`<=`), or greater than or equal (`>=`) another value.

- C provides logical operators for combining relational expressions: `&&` ANDs the results of the expressions and `||` ORs them. Both make use of short-circuit evaluation. There is one other logical operator, `!`, that inverts the logical value of an expression.

- C provides operators for shifting bits (`<<` and `>>`), inverting bits (`~`), and doing logical operations on bits (`&`, `|`, and `^`).

- C provides a set of shorthand assignment operators. There are both arithmetic shorthand operators (`+=`, `-=`, `*=`, `/=`, and `%=`) and bitwise operators (`&=`, `|=`, `<<=`, `>>=`, and `^=`). These operators often lead to more readable and efficient code.

CHAPTER SUMMARY

- C provides increment (**++**) and decrement (**--**) operators, in both prefix and postfix forms. As prefix operators, the operation happens before the value is used. As postfix operators, the operation happens after the value is used.

- C provides a conditional operator (`?:`) for selecting between a pair of expressions, a comma operator (`,`) for sequentially evaluating a list of expressions, and a **`sizeof`** operator for determining the number of bytes in a variable or type.