CHAPTER 15

THE PREPROCESSOR

15.1 PREPROCESSOR DIRECTIVES

15.2 SIMPLE MACRO SUBSTITUTION

- Syntactic Replacement

15.3 MACRO SUBSTITUTION WITH PARAMETERS

- Using Macros in Macro Definitions
- Some Useful Macros
- Potential Problems
- Macro Operators
- Undefining a Name

15.4 FILE INCLUSION

- Including Data Type Definitions

15.5 CONDITIONAL COMPILATION

- Other Testing Directives
- Predefined Names

15.6 OTHER PREPROCESSOR DIRECTIVES

15.7 CASE STUDY—IMPLEMENTING SETS WITH MACROS

PREPROCESSOR DIRECTIVES

DIRECTIVE	USE
`#define`	define a macro
`#include`	include text from a file
`#undef`	undefine a macro
`#if`	test if a compile-time condition holds
`#else`	indicate alternatives if a test fails
`#elif`	combine **`#else`** and **`#if`**
`#endif`	end a preprocessor conditional (**`#if, #ifdef, #ifndef`**)
`#ifdef`	test if a symbol is defined
`#ifndef`	test if a symbol is not defined
`#line`	provide line number (and optional file name) for compiler messages
`#error`	terminate processing early
`#pragma`	implementation-dependent directive

comstr.h

(Figure 15.1 – Page 373)

```
/*
 * Generally useful, commonly used string definitions.
 */
#define DIGITS    "0123456789"
#define LOWERS    "abcdefghijklmnopqrstuvwxyz"
#define UPPERS    "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
#define PUNCTS    ",.;:!?"
#define QUOTES    "`'\""
#define BRACKETS  "(){}[]"
```

cntchrs.c

(Figure 15.2 – Page 374)

```
/*
 * Count various types of characters.  Extension of the version
 * found in Chapter 10 that now uses defined string constants.
 */
#include <stdio.h>
#include <string.h>
#include <stddef.h>
#include "comstr.h"                     /* for string constants */

typedef unsigned long COUNTER;

const int PrintWidth = 11;              /* counter field size */

int main()
{
  int     c;
  COUNTER quotes   = 0,
          brackets = 0,
          delims   = 0,
          digits   = 0,
          lowers   = 0,
          uppers   = 0,
          others   = 0;
```

cntchrs.c

(Figure 15.2 – Page 374)

```
  while ((c = getchar()) != EOF)
    if (strchr(DIGITS, c) != NULL)
      digits++;
    else if (strchr(LOWERS, c) != NULL)
      lowers++;
    else if (strchr(UPPERS, c) != NULL)
      uppers++;
    else if (strchr(QUOTES, c) != NULL)
      quotes++;
    else if (strchr(BRACKETS, c) != NULL)
      brackets++;
    else if (strchr(PUNCTS, c) != NULL)
      delims++;
    else
      others++;

  printf("Quotes     %*lu\n", PrintWidth, quotes);
  printf("Brackets   %*lu\n", PrintWidth, brackets);
  printf("Delimiters %*lu\n", PrintWidth, delims);
  printf("Digits     %*lu\n", PrintWidth, digits);
  printf("Lowercase  %*lu\n", PrintWidth, lowers);
  printf("Uppercase  %*lu\n", PrintWidth, uppers);
  printf("Others     %*lu\n", PrintWidth, others);

  return 0;
}
```

circumf1.c

(Figure 15.3 – Page 375)

```
/*
 * Compute the circumference of a circle.
 */
#include <stdio.h>

#define TWOPI (3.1415926 * 2.0)

int main()
{
  double radius, circumf;

  printf("Enter radius: ");
  scanf("%lf", &radius);
  circumf = TWOPI * radius;
  printf("Circumference: %g\n", circumf);
  return 0;
}
```

circumf2.c

(Figure 15.4 – Page 375)

```
/*
 * Compute the circumference of a circle.
 */
#include <stdio.h>

#define PI    3.1415926
#define TWOPI (PI + PI)

int main()
{
  double radius, circumf;

  printf("Enter radius: ");
  scanf("%lf", &radius);
  circumf = TWOPI * radius;
  printf("Circumference: %g\n", circumf);
  return 0;
}
```

yesorno.c

(Figure 15.5 – Page 377)

```
/*
 * Program to print a prompt and get yes/no answer from user.
 */
#include <stdio.h>
#include <ctype.h>

#define FOREVER   for (;;)                  /* infinite loop */
#define IS        ==                        /* equality test */
#define ISNOT     !=                        /* inequality test */

int main()
{
  int yesorno(void);

  printf("Enter YES or NO answer: ");
  printf("That was a %s!\n", (yesorno()) ? "YES" : "NO");
  return 0;
}

/* Get a YES/NO answer from user */

int yesorno(void)
{
  int c;                                    /* input character */

  FOREVER
  {
    if ((c = tolower(getchar())) IS EOF || c IS 'y' || c IS 'n')
      return c IS 'y';
    while (c ISNOT '\n' && c ISNOT EOF)
      c = getchar();                        /* skip rest of line */
    printf("Please answer with a YES or NO: ");
  }
}
```

pascal.h

(Figure 15.6 – Page 378)

```
/*
 * Definitions for Pascal-like keywords
 */
#define INTEGER    int
#define REAL       double
#define IF         if (
#define THEN       )
#define ELSE       else
#define WHILE      while (
#define DO         )
#define BEGIN      {
#define END        }
#define PROCEDURE  void
#define FUNCTION
#define RETURN     return
```

power.c

(Figure 15.7 – Page 378)

```
/*
 * A function to compute powers using Pascal-like keywords.
 */
#include <stdio.h>
#include "pascal.h"                        /* for Pascal keywords */

FUNCTION REAL power(REAL x, INTEGER exp)
BEGIN
  REAL p = 1;

  IF exp > 0 THEN
    WHILE exp-- > 0 DO p *= x;
  ELSE
    WHILE exp++ < 0 DO p /= x;
  RETURN p;
END
```

bits.h

(Figure 15.8 – Page 379)

```
/*
 * Macro replacements for our earlier bit-accessing functions.
 */
#define GETBIT(w,n)  (((unsigned int) (w) >> (n)) & 01)
#define SETBIT(w,n,v) \
        ((v == 0) ? ((unsigned int) (w) & ~(01 << (n))) \
                  : ((unsigned int) (w) |  (01 << (n))))

#define GETBITS(w,n,k) \
        (((unsigned int) (w) & (~(~0 << (k)) << (n))) >> (n))
#define SETBITS(w,n,k,v) \
        (((unsigned int) (w) & ~(~(~0 << (k)) << (n))) | ((v) << (n)))
```

usebits.c

(Figure 15.9 – Page 381)

```
/*
 * Revised program to pack employee information into a single word.
 * Now uses bit macros rather than bit functions.
 */
#include <stdio.h>
#include "bits.h"

enum status {SINGLE, MARRIED, SEPARATED, DIVORCED};
enum genders {MALE, FEMALE};

#define MSBIT     0       /* marital status: bits 0-1 */
#define MSBITS    2
#define AGEBIT    2       /* age: bits 2-8 */
#define AGEBITS   7
#define SEXBIT    9       /* sex: bit 9 */
#define YRSBIT    10      /* year: bits 10-15 */
#define YRSBITS   6
```

usebits.c

(Figure 15.9 – Page 381)

```
int main()
{
  enum status  mstat;
  enum genders sex;
  unsigned int age, years;
  unsigned int info = 0;       /* holds info on one person */

  printf("male=%i, female=%i? ", MALE, FEMALE);
  scanf("%i", &sex);
  info = SETBIT(info, SEXBIT, sex);

  printf("age? ");
  scanf("%i", &age);
  info = SETBITS(info, AGEBIT, AGEBITS, age);

  printf("single=%i, married=%i, separated=%i, divorced=%i? ",
         SINGLE, MARRIED, SEPARATED, DIVORCED);
  scanf("%i", &mstat);
  info = SETBITS(info, MSBIT, MSBITS, mstat);

  printf("years employed? ");
  scanf("%i", &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;
}
```

bits2.h

(Figure 15.10 – Page 382)

```
/*
 * New version of our bit macros built up from simpler macros.
 *   UINT(x)          - x cast to unsigned int.
 *   ONBIT(n)         - only bit n on.
 *   OFFBIT(n)        - only bit n off.
 *   ONBITS(n,k)      - only bits n thru n+k on.
 *   OFFBITS(n,k)     - only bits n thru n+k off.
 *   SETBITON(w,n)    - turn bit n on in w.
 *   SETBITOFF(w,n)   - turn bit n off in w.
 *   GETBIT(w,n)      - value of bit n in  w.
 *   GETBITS(w,n,k)   - value of bits n thru n+k in w.
 *   SETBIT(w,n,v)    - give bit n value v in unsigned w.
 *   SETBITS(w,n,k,v) - give bits n thru n+k value v in w.
 */
#define UINT(x)          ((unsigned int) (x))

#define ONBIT(n)         (01 << (n))
#define OFFBIT(n)        (~ONBIT(n))
#define ONBITS(n,k)      (~(~0 << (k)) << (n))
#define OFFBITS(n,k)     (~ONBITS(n,k))

#define SETBITON(w,n)    (UINT(w) | ONBIT(n))
#define SETBITOFF(w,n)   (UINT(w) & OFFBIT(n))

#define GETBIT(w,n)      ((UINT(w) >> (n)) & 01)
#define SETBIT(w,n,v)    (((v) == 0) ? SETBITOFF(w,n) : SETBITON(w,n))
#define GETBITS(w,n,k)   ((UINT(w) & ONBITS(n,k)) >> (n))
#define SETBITS(w,n,k,v) ((UINT(w) & OFFBITS(n,k)) | ((v) << (n)))
```

inrange.c

(Figure 15.11 – Page 383)

```
/*
 * A version of our range-checking program that uses a macro.
 */
#include <stdio.h>

#define INRANGE(x,y,v)  ((v) > (x) && (v) <= (y))

#define MAXINT   1000
#define MININT      0
#define MAXDBL   9999.99
#define MINDBL  -9999.99

int main()
{
  double d;
  int    i;

  printf("Enter values: ");
  scanf("%i %lf", &i, &d);
  printf("%i is %s range!\n",
         i, INRANGE(MININT, MAXINT, i) ? "in" : "out of");
  printf("%f is %s range!\n",
         d, INRANGE(MINDBL, MAXDBL, d) ? "in" : "out of");
  return 0;
}
```

allocs.h

(Figure 15.12 – Page 383)

```
/*
 * Useful macros for dynamically allocating things.
 */
#include <stdlib.h>
#include <stddef.h>
#include <string.h>

#define ALLOC(n,t)        (malloc((n) * sizeof(t)))
#define COPY(np,op,n,t)   (memcpy(np,op,(n) * sizeof(t)))
#define ZERO(p,n,t)       (memset(p,(n) * sizeof(t),0))

#define TABLEDUP(np,op,n,t) \
        (((np = ALLOC(n,t)) != NULL) ? COPY(np,op,n,t) : NULL)

#define STRALLOC(s)       (malloc(strlen(s) + 1))
#define STRDUP(ns,os)     ((ns = STRALLOC(os)) ? strcpy(ns,os) : NULL)
```

dup.c

(Figure 15.13 – Page 385)

```
/*
 * A program using our memory allocation macros.
 */
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include "allocs.h"

int main()
{
  int   table[] = {10, 87, 95, 89, 15, 45, 67, 22, 79, 32};
  char  *string = "abcdefghi";
  const int n  =  sizeof(table)/sizeof(table[0]);
  int   *newtable;
  char  *newstr;
  int   i;

  if (TABLEDUP(newtable, table, n, int) == NULL)
    printf("Integer table copy failed...\n");
  else
  {
    ZERO(table, n, int);
    for (i = 0; i < n; i++)              /* display copied array */
      printf("%i\n", newtable[i]);
  }
  if (STRDUP(newstr, string) == NULL)
    printf("String copy failed...\n");
  else
  {
    int len=strlen(string);

    ZERO(string, len, char);
    printf("%s\n", newstr);
  }
  return 0;
}
```

Sample Output – debugex

```
i=69
DEBUG: i (INTEGER) = 69
DEBUG: d (DOUBLE) = 3.141593 (3.141593e+00)
```

debugex.c

(Figure 15.14 – Page 387)

```
/*
 * Using the preprocessor to obtain readable debugging output.
 */
#include <stdio.h>

#define DUMPINT(x)     printf(#x "=%i\n", x)
#define DISPLAY(x, t)  print ## t (#x, x)

int main()
{
  void   printint(char *name, int x);
  void   printdouble(char *name, double x);
  int    i = 69;
  double d = 3.1415926;

  DUMPINT(i);
  DISPLAY(i, int);  DISPLAY(d, double);
  return 0;
}

void printint(char *name, int x)
{
  printf("DEBUG: %s (INTEGER) = %i\n", name, x);
}

void printdouble(char *name, double x)
{
  printf("DEBUG: %s (DOUBLE) = %f (%e)\n", name, x, x);
}
```

defs.h

(Figure 15.15 – Page 389)

```
/*
 * Our default definitions.
 */
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define FOREVER for(;;)
#define IS      ==
#define ISNOT   !=

#define MIN(x,y)        ((x) < (y) ? (x) : (y))    /* smaller */
#define MAX(x,y)        ((x) < (y) ? (y) : (x))    /* larger */
#define INRANGE(x,y,v)  ((v) >= (x) && (v) <= (y))
#define STREQ(x,y)      (strcmp(x,y) == 0)         /* hide */
#define STRLT(x,y)      (strcmp(x,y) < 0)          /*  strcmp */
#define STRGT(x,y)      (strcmp(x,y) > 0))         /*  ugliness */
#define PRINT_STRING(s) (printf("%s\n", s))        /* print string */
```

boolean.h

(Figure 15.16 – Page 390)

```
/*
 * Header file defining BOOLEAN data type.
 */
#if !defined(BOOLEAN)

#define     BOOLEAN         short
#define     TRUE            1
#define     FALSE           0
#define     NOT             !
#define     AND             &&
#define     OR              ||
#define     BOOLSTR(x)      (x) ? "TRUE" : "FALSE"

#endif
```

uniq.c

(Figure 15.17 – Page 390)

```
/*
 * A new version of our program to strip duplicate lines.
 */
#include "defs.h"                   /* Our definitions */
#include "boolean.h"                /* Our Boolean type */

#define MAXLEN  80                  /* longest line length */

int main()
{
  int getline(char *buf, int size);
  char    curr[MAXLEN + 1],         /* current line */
          prev[MAXLEN + 1];         /* previous line */
  BOOLEAN first = TRUE;             /* first time through? */

  for (; getline(curr, MAXLEN) ISNOT -1; strcpy(prev, curr))
    if (first OR !STREQ(prev, curr))
    {
      PRINT_STRING(curr);
      first = FALSE;
    }
  return 0;
}
```

Sample Output – average1

```
Averaging 5 values.
ptr=68fab31c, endptr=68fab330
*ptr=91, sum=91
*ptr=40, sum=131
*ptr=78, sum=209
*ptr=23, sum=232
*ptr=56, sum=288
57.600000
```

average1.c

(Figure 15.18 – Page 392)

```
/*
 * Computing the average of an array, with debugging code.
 */
#if defined(DEBUG)
#include <stdio.h>
#endif

double table_average(const int a[], int n)
{
  long      sum = 0L;
  const int *ptr = a;                /* traversing pointer */
  const int * const endptr = a + n;  /* pointer to just past the end */

#if defined(DEBUG)
  printf("ptr=%p, endptr=%p\n", (void *) ptr, (void *) endptr);
#endif

  for ( ; ptr < endptr; ptr++)
#if defined(DEBUG)
  {
    sum += *ptr;
    printf("*ptr=%i, sum=%li\n", *ptr, sum);
  }
#else
    sum += *ptr;
#endif

  return (n != 0) ? (double) sum / n : 0.0;
}
```

avgmain.c

(Figure 15.19 – Page 393)

```
/*
 * Using our table average function.
 */
#include <stdio.h>

int main()
{
  double table_average(const int a[], int n);
  const int table[] = {91, 40, 78, 23, 56};
  const int n       = sizeof(table)/sizeof(table[0]);

#if defined(DEBUG)
  printf("Averaging %i values.\n", n);
#endif
  printf("%f\n", table_average(table, n));
  return 0;
}
```

average2.c

(Figure 15.20 – Page 394)

```
/*
 * Computing the average of an array, with debugging code.
 */
#if DEBUG > 0
#include <stdio.h>
#endif

double table_average(const int a[], int n)
{
  long      sum = 0L;
  const int *ptr = a;                /* traversing pointer */
  const int * const endptr = a + n;  /* pointer to just past the end */

#if DEBUG > 0
  printf("ptr=%p, endptr=%p\n", (void *) ptr, (void *) endptr);
#endif

  for ( ; ptr < endptr; ptr++)
#if DEBUG > 1
  {
    sum += *ptr;
    printf("*ptr=%i, sum=%li\n", *ptr, sum);
  }
#else
    sum += *ptr;
#endif

  return (n != 0) ? (double) sum / n : 0.0;
}
```

PREDEFINED NAMES

NAME	VALUE OR TEST
__LINE__	current line number (an integer)
__FILE__	current file name (a string constant)
__DATE__	date source file is translated (string of the form: MMMDDYYYY)
__TIME__	time source file is translated (string of the form: HHMMSS)
__STDC__	defined if ANSI-compatible

assert.h

(Figure 15.21 – Page 396)

```
/*
 * Our version of the ASSERT macro.
 */
#include <stdlib.h>

#define ASSERT(cond) \
  ((cond) || \
   printf("Assertion (" # cond ") failed (%s,%i)\n", \
          __FILE__, __LINE__), exit(1))
```

Sample Output – average3

```
Assertion (ptr != NULL) failed (average3.c,13)
```

average3.c

(Figure 15.22 – Page 397)

```
/*
 * Computes the average of an array.  Contains an assertion.
 */
#include <stdio.h>
#include "assert.h"

double table_average(const int a[], int n)
{
  long      sum = 0L;
  const int *ptr = a;                /* traversing pointer */
  const int *const endptr = a + n;   /* pointer to just past the end */

  ASSERT(ptr != NULL);
  for ( ; ptr < endptr; ptr++)
    sum += *ptr;
  return (n != 0) ? (double) sum / n : 0.0;
}
```

sets.h

(Figure 15.23 – Page 399)

```
/*
 * Macro package for dynamically allocated sets.
 */
#include <stdlib.h>

typedef void *SET;
typedef unsigned short USHORT;

#define US_BITS 16                /* Bits in unsigned short */

/* Utility macros:
 *   WORD:  index of word holding element
 *   BIT:   index of bit holding element
 *   WORDS: number of words needed to hold entire set
 */
#define WORD(e)         ((e) / US_BITS)
#define BIT(e)          ((e) % US_BITS)
#define WORDS(n)        (((n) / US_BITS) + (((n) % US_BITS) != 0))
/*
 * Macros meant for the outside world:
 *   SET_CREATE:  form a new set of n elements.
 *   SET_DESTROY: get rid of existing set.
 *   SET_ADD:     add element to an existing set.
 *   SET_MEMBER:  is item in set?
 *   SET_SWITCH:  move item from one set to another.
 */
#define SET_CREATE(n)   (calloc(WORDS(n),sizeof(USHORT)))
#define SET_DESTROY(s)  free((s))
#define SET_ADD(s,e)    (* ((USHORT *) (s) + WORD(e)) |= 1 << BIT(e))
#define SET_DELETE(s,e) (* ((USHORT *) (s) + WORD(e)) &= ~(1 << BIT(e)))
#define SET_MEMBER(s,e) \
          (((unsigned) (* ((USHORT *) (s) + WORD(e))) >> BIT(e)) & 01)
#define SET_SWITCH(n,o,v)   (SET_DELETE(o,v), SET_ADD(n,v))
```

usesets.c

(Figure 15.24 – Page 400)

```
/*
 * Identify duplicates in the input.
 */
#include "defs.h"
#include "sets.h"               /* for SETs */

int main()
{
  void      set_print(const char *name, const SET s, int max);
  const int MAXELEMS = 512;                  /* # of set elements */
  SET       unique = SET_CREATE(MAXELEMS),   /* unique and */
            dup    = SET_CREATE(MAXELEMS);   /*   duplicate sets */
  int       r;                               /* scanf return */
  int       inp;                             /* input value */

  while ((r = scanf("%i", &inp)) ISNOT EOF)
  {
    if (r ISNOT 1)
    {
      printf("Value in error.\n");
      break;
    }
    if (!INRANGE(0, MAXELEMS - 1, inp))
    {
      printf("Value %i not between %i and %i\n", inp, 0, MAXELEMS - 1);
      continue;
    }
    if (SET_MEMBER(unique, inp))
      SET_SWITCH(dup, unique, inp);
    else if (!SET_MEMBER(dup, inp))
      SET_ADD(unique, inp);
  }
  set_print("Unique", unique, MAXELEMS);
  set_print("Duplicate", dup, MAXELEMS);
  return 0;
}
```

usesets.c

(Figure 15.24 – Page 400)

```
void set_print(const char *name, const SET set, int max)
{
  int i;                          /* next potential element */

  printf("%s values\n", name);
  for (i = 0; i < max; i++)
    if (SET_MEMBER(set, i))
      printf("%i\n", i);
}
```

REMINDERS

- To avoid problems, always parenthesize the replacement text.
- Remember that the preprocessor doesn't know C—it blindly replaces names with replacement text.
- Use expressions as macro arguments sparingly, and avoid them entirely when they contain side effects.
- Make sure to parenthesize all occurrences of the parameter inside the macro's replacement text.

CHAPTER SUMMARY

- All C programs are first passed through a preprocessor before the compiler ever sees them.

- **`#define`** is a general mechanism for having the preprocessor substitute text for names.

- We can define names with parameters, so that the replacement text becomes a template filled in when we use the name.

- We can use **`#undef`** to stop substituting text for a name.

- **`#include`** has the preprocessor substitute the contents of the named file, which can be either a system header file or one of our own.

- **`#if`** (together with **`#else`** and **`#endif`**) allows us to pass different code to the compiler depending on whether a particular name is defined at compile time or what value it has.