CHAPTER 21

EFFICIENCY

21.1 SOME BASICS

21.2 COMMON SOURCES OF INEFFICIENCY

- Unnecessary Conversions
- Unnecessary Arithmetic

21.3 C EFFICIENCY AIDS

- Compile-Time Initialization
- Using Pointers
- Using Macros

21.4 SHRINKING SPACE REQUIREMENTS

- Eliminating Unused Storage
- Eliminating Extra Information

21.5 CASE STUDY—REPLACING LIBRARY ROUTINES

- Managing Our Own Free Storage Pool
- Writing Our Own Input Handler

avg.c

(Figure 21.1 – Page 535)

```
/*
 * Compute average of an array, improved pointer version.
 */
double table_average(int *ptr, int n)
{
  long sum     = 0L;
  int  *endptr = ptr + n;

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

pow.c

(Figure 21.2 – Page 536)

```
/*
 * Carefully avoiding automatic conversions when computing exponents.
 */
#include <stdio.h>
#include <math.h>

int main()
{
  printf("10^6 = %g\n", pow(10.0, 6.0));
  printf("10^-2 = %g\n", pow(10.0, -2.0));
  printf("10^3 = %g\n", pow(10.0, 3.0));
  return 0;
}
```

avg2.c

(Figure 21.3 – Page 538)

```
/*
 * Compute average value in a table of integers.
 */
double table_average(int a[], int n)
{
  int *ptr = a;
  long sum = 0L;

  while (ptr < a + n)
    sum += *ptr++;
  return (n != 0) ? (double) sum / n : 0.0;
}
```

fastpow.c

(Figure 21.4 – Page 540)

```
/*
 * Compute powers using table of commonly accessed powers.
 */
#include <stdio.h>
#include <math.h>

#define TWO_TO_THE(x)   (1L << (x))
#define FOUR_TO_THE(x)  (1L << ((x) << 1))

#define MAXTHREES 7

long power(int x, int y)
{
  static long threes[] = { 1L, 3L, 9L, 27L, 81L, 243L, 729L};

  switch(x)                              /* special cases */
  {
    case 0: return 0L;
    case 1: return 1L;
    case 2: return TWO_TO_THE(y);
    case 3: if (y >= 0 && y <= MAXTHREES)
              return threes[y];
            break;
    case 4: return FOUR_TO_THE(y);
  }
  return pow(x,y);
}

int main()                               /*  test program */
{
  printf("2^5: %li\n", power(2,5));
  printf("4^1: %li\n", power(4,1));
  printf("7^3: %li\n", power(7,3));
  printf("3^6: %li\n", power(3,6));
  printf("3^12: %li\n", power(3,12));
  return 0;
}
```

search.c

(Figure 21.5 – Page 541)

```
/*
 * Driver program for search routines.
 */
#include <stdio.h>

int main(int argc, char *argv[])
{
  int *table_search(int [], int, int);
  int value, i;
  int table[11] = {10, 24, 7, 34, 87, 56, 45, 1, 9, 20};
  int *ptr;

  for (i = 1; i < argc; i++)
    if (sscanf(argv[i], "%i", &value) == 1)
      if ((ptr = table_search(table, 10, value)) == NULL)
        printf("Value %i not found\n", value);
      else
        printf("Value %i found at %i\n", value, ptr - table);
  return 0;
}
```

searcha.c

(Figure 21.6 – Page 541)

```
/*
 * New linear search function (assumes extra array element).
 * Array subscripting version.
 */
#include <stddef.h>

int *table_search(int a[], int num, int target)
{
  int i;

  for (a[num] = target, i = 0; a[i] != target; i++)
    ;
  return (i < num) ? &a[i] : NULL;
}
```

searchb.c

(Figure 21.7 – Page 542)

```
/*
 * New linear search function (assumes extra array element).
 * Pointer indexing version.
 */
#include <stddef.h>

int *table_search(int *ptr, int num, int target)
{
  int *endptr = ptr + num;

  for (*endptr = target; *ptr != target; ptr++)
    ;
  return (ptr < endptr) ? ptr : NULL;
}
```

searchc.c

(Figure 21.8 – Page 542)

```
/*
 * New linear search function (assumes extra array element).
 * More compact, pointer indexing version.
 */
#include <stddef.h>

int *table_search(int *ptr, int num, int target)
{
  int *endptr = ptr + num;

  *endptr = target;
  while (*ptr++ != target)
    ;
  return (--ptr < endptr) ? ptr : NULL;
}
```

round.c

(Figure 21.9 – Page 543)

```
/*
 * Defining and using ROUND and TRUNC macros.
 */
#include <stdio.h>

#define TRUNC(x) ((int) (x))
#define ROUND(x) ((int) (x + 0.5))

int main()
{
  double x;

  scanf("%lf", &x);
  printf("value=%f,rounded=%i,truncated=%f\n", x, ROUND(x), TRUNC(x));
  return 0;
}
```

date.h

(Figure 21.10 – Page 545)

```
/*
 * Date header file.
 */
typedef struct date
{
  unsigned int day : 4;
  unsigned int month : 5;
  unsigned int year : 16;
}
  DATE;

int date_equal(const DATE *d1ptr, const DATE *d2ptr);
int date_lt(const DATE *d1ptr, const DATE *d2ptr);
int date_gt(const DATE *d1ptr, const DATE *d2ptr);
```

fsp.h

(Figure 21.11 – Page 547)

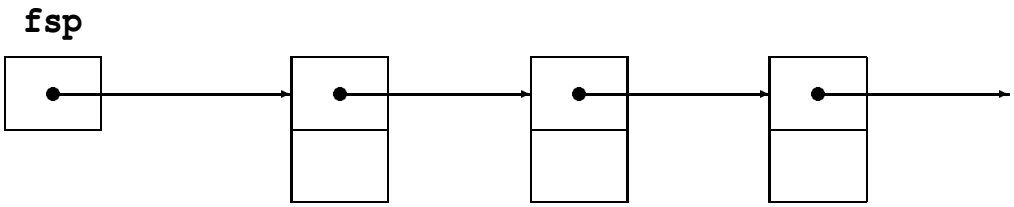
```
/*
 * Prototypes for routines that manage free storage pool.
 */
#include "lnodes.h"

LISTNODE *fsp_alloc(void);
void fsp_free(LISTNODE *ptr);
```

fsp.c

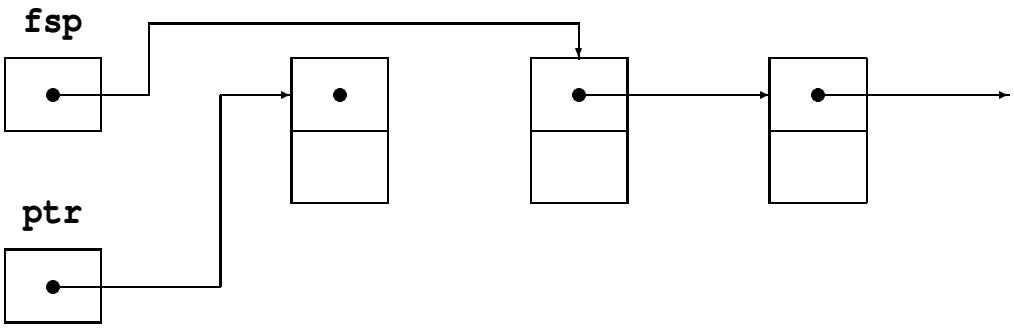
(Figure 21.12 – Page 547)

```
/*
 * Managing our own free storage pool.
 */
#include  <stddef.h>
#include  "fsp.h"

static const int nodes = 100;     /* allocate 100 nodes at a time */
static LISTNODE *fsp = NULL;      /* our free storage pool */

LISTNODE *fsp_alloc(void)
{
  LISTNODE *nptr = NULL;          /* pointer to new node */
  int      i = 0;

  if (fsp == NULL)                /* make initial pool */
    for (; i < nodes && (nptr = malloc(sizeof(LISTNODE))) != NULL; i++)
    {
      nptr->nextptr = fsp;
      fsp = nptr;
    }
  nptr = fsp;
  if (fsp != NULL)                /* pool node from pool */
  {
    fsp = fsp->nextptr;
    nptr->nextptr = NULL;         /* sever connection w/pool */
  }
  return nptr;
}

void fsp_free(LISTNODE *ptr)
{
  ptr->nextptr = fsp;             /* assume legitimate pointer */
  fsp = ptr;
}
```

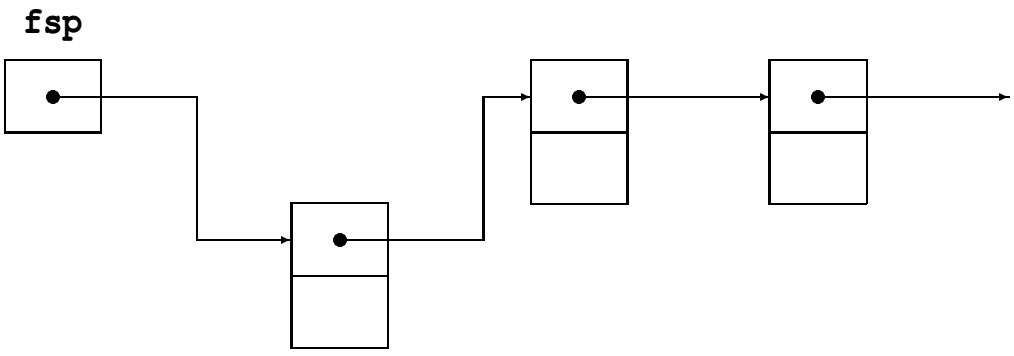
A FREE STORAGE POOL



(a) Nodes in free storage pool.



(b) Allocating a node from the front of the list.



(c) Returning a node to the front of the list.

lnodes.c

(Figure 21.14 – Page 549)

```
/*
 * Manage linked list nodes using free storage pool.
 *   ln_create - create new list node.
 *   ln_destroy - free space used by existing list node.
 */
#include "fsp.h"

LISTNODE *ln_create(void *dptr, LISTNODE *lptr)
{
  LISTNODE *nptr = fsp_alloc();

  if (nptr != NULL)
  {
    nptr->dataptr = dptr;                /* points to data */
    nptr->nextptr = lptr;                /* points to next node */
  }
  return nptr;
}

void ln_destroy(LISTNODE *nptr)
{
  fsp_free(nptr);                        /* destroy node, not data */
}
```

tabfill.c

(Figure 21.15 – Page 550)

```
/*
 * Read an array of positive integers without scanf.  Assumes one
 * single integer per input line.
 */
#include <stdio.h>
#include <ctype.h>

#define MAX 100

int main()
{
  int table_fill(int table[], int max);
  int table[MAX];

  printf("Read %i values\n", table_fill(table,MAX));
  return 0;
}
```

tabfill.c

(Figure 21.15 – Page 550)

```
int table_fill(int table[], int max)
{
  int *ptr    = table;             /* ptr to next array element */
  int *endptr = ptr + max;         /* ptr to last array element */
  int c;                           /* next character */

  for (*ptr = 0; ptr < endptr; ptr++)
  {
    while (!isdigit(c = getchar()) && c != '\n' && c != EOF)
      ;                            /* find a digit */

    if (isdigit(c))
      do                           /* translate number */
        *ptr = *ptr * 10 + (c - '0');
      while (isdigit(c = getchar()));
    else
      *ptr = -1;                   /* -1 indicates bad number */

    while (c != '\n' && c != EOF)  /* skip rest of line */
      c = getchar();

    if (c == EOF)                  /* reached EOF */
      break;
  }

  return ptr - table;              /* # of values read */
}
```

REMINDERS

- Avoid unnecessary conversions.
- Avoid unnecessary arithmetic.

CHAPTER SUMMARY

- Choose appropriate algorithms and data structures for your task and then forget about efficiency until after you get your program running and it becomes clear that it runs too slowly.

- Before you start trying to make your program more efficient, profile your program to determine which functions are using up the most time and need to be improved. Then concentrate all your energy on the few functions that are the worst offenders.

- Two common sources of inefficiency are unnecessary conversions and unnecessary arithmetic. If at all possible, make sure any expression or assignment involves only a single type. And if you have to do arithmetic, try to do integer arithmetic rather than floating point.

- Use pointer indexing rather than array subscripting wherever possible, especially when sequentially accessing elements within higher-dimensional arrays.

- Function calls are another source of inefficiency. For small, frequently called functions, we can usually obtain a substantial speedup by turning them into macros.

CHAPTER SUMMARY

- One general technique for speeding up programs is to replace general-purpose routines with specialized versions tuned to the task at hand.

- Two techniques for making programs use less space are eliminating unused storage and choosing the smallest appropriate data type.