CZ1106

Lecturenote (Part 2)

Chapter 3

Bit Manipulation

1. Introduction

- C allows programs to manipulate data at bit level.
- In order to speed operations, bits are organized into groups such as a *byte,* which is normally eight bits, or a *word* containing several bytes.
- Word sizes vary from machine to machine; on commercially available computers they range from 16 bits at the lower end to 64 bits on some large, scientifically oriented machines.
- In C an unsigned int would normally corresponds to a machine word and is the most natural type to use if bits are being manipulated, although signed int is sometimes used.

2. Basic Operations

The operations available on words considered as bit values are the logical operations and various types of shift. The logical operations are:

& Bitwise AND

Each bit of the left-hand operand is logically ANDed with each bit of the right-hand operand. ANDing two bits together gives the result zero unless both bits are one.

(both must be one to get one)

&	0	1
0	0	0
1	0	1

The & Table

E.g.,

```
unsigned i = 269, j = 187, k;

k = i & j;
printf("k = %u",k);
```

will give **k = 9**

```
   .....00100001101  (269)
& .....00010111011  (187)
   .....00000001001  (  9)
```

| Bitwise inclusive OR

Each bit of the left-hand operand is logically ORed with each bit of the right-hand operand. Inclusively ORing two bits together gives the result one if either of the two bits is one, or both bits are one.

(either one or both to get one)

|	0	1
0	0	1
1	1	1

The | Table

E.g.,

```
unsigned i = 269, j = 187, k;

k = i | j;
printf("k = %u",k);
```

will give **k = 447**

```
  .......00100001101  (269)
| .......00010111011  (187)
  .......00110111111  (447)
```


^ Bitwise exclusive OR

Each bit of the left-hand operand is exclusively ORed with each bit of the right-hand operand. Exclusively ORing two bits together gives the result one if either, but not both, of the two bits is one.

(one and only one to get one)

^	0	1
0	0	1
1	1	0

The ^ Table

E.g.,

```
unsigned i = 269, j = 187, k;

k = i ^ j;
printf("k = %u",k);
```

will give **k = 438**

```
  .......00100001101  (269)
^ .......00010111011  (187)
  .......00110110110  (438)
```

~ Bitwise complement (also called NOT)

This is a monadic operator (i.e., has no left-hand operand) and it reverses each bit of its operand.

E.g.,

```
unsigned i = 269;

i = ~ i;
printf("i = %u",i);
```

will give **i = 65266** on a 16-bit machine.

```
 ~ 0000 0001  0000     1101  (269)
   1111 1110  1111     0010  (65266)
```


<< Left shift operator

E.g., target << n, where target and n can be expressions. n must be a positive value. Bit contents will be dropped when shifted to the left end.

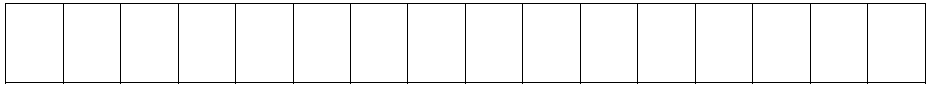
E.g.,

```
unsigned i = 19, j;

j = i << 2;
      /* meaning that shift the bit pattern of i to the
         left by 2 places and assign the result to j */

printf("%u",j);
```

will give **76**

```
...00010011 << 2  = ...01001100
                  = 76
```

Take note that shift left 2 places is equivalent to "multiplied by 4". What is the result after shifting left 5 places ?



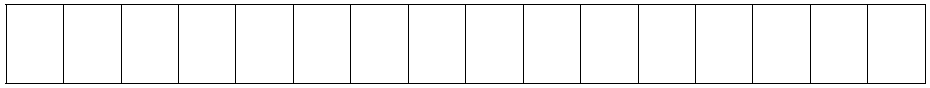
>> Right shift operator

E.g.,

```
unsigned i = 110;

i = i >> 3;
  /* means to shift the contents of bit pattern
     in i to the right by 3 places and assign
     the result back to i */

printf("%u",i);
```

will give **13**

```
...1101110= ...1101 (some bits on the right
                     hand side are dropped)
          = 13
```



Take note that shift right 3 places is equivalent to "divided by 8". What is the result after shifting right 5 places ?

Short Form :

. a = a+b; can be written as a += b;
. Similarly a = a|b; can be written as a|=b;
. Other operators include &=, ^=, <<= and >>=.

Bitwise Operations :

E.g.,

Let a = 1101 1100 0101 1100 in binary. To extract the rightmost eight bits of a, we use the & operator as follows:

```
     1101 1100 0101 1100
&    0000 0000 1111 1111
_____________________________
     0000 0000 0101 1100
```

Same technique can be used to test individual bits or groups of bits.

E.g., to test the right-most bit of V

if (v & 1) ...

```
     1101 1100 0101 110?
&    0000 0000 0000 0001
_____________________________
     0000 0000 0000 000?
```

E.g., to test the right-most 3 bits of V

if (v & 7) ...

```
     1101 1100 0101 1???
&    0000 0000 0000 0111
_____________________________
     0000 0000 0000 0???
```

Application of & Operator :

All odd numbers have a 1 in the right-most bit of its binary representation. So, to check if a number is odd we simply check its right-most bit. The following function will return 0 (false) if its parameter is an even number and 1 (true) if it is odd.

```
int odd(int n)
{
   return(n & 1);      /* Test if n is odd */
}
```

To make use of this function to check if num is odd :

```
int num;
      :
      :
if odd (num)  /* call the above function */
   printf ("num is odd");
else
   printf ("num is even");
```

Application of | Operator :

OR operations can be used to set bits in a word. For instance, to set the leftmost four bits of a 16-bit integer v to 1, leaving the others unchanged, we can use the expression v |= 0xf000.

E.g., v = 0x639c;

```
    0110  0011 1001 1100   (0x639c)
|   1111  0000  0000 0000   (0xf000)
-------------------------------------
    1111  0011 1001 1100   (0xf39c)
```


3. An Efficient Way to Perform Bit Counting

. A simple practical example involves trying to count the number of bits in a word which are set to 1, a problem which can arise in data communication (see Practical 2).

. The obvious way is to check every bit. This is implemented by a static loop instruction.

Program

```
/* AC2-1.C */

#include<stdio.h>

int Count_Bits_Obvious(unsigned n)
{
  int count=0,i;
  int mask=1;

  for (i=0;i<16;i++)
  {
    if (n & mask)  count++;
    n>>=1;
  }

  return count;
}
```

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

```
main()
{
  int number;

  printf("Enter an integer:");
  scanf("%d",&number);
  printf("There are %d 1-bits.\n",
                        Count_Bits_Obvious(number));
  return 0;
}
```

Screen Output

Enter an integer:*57*
There are 4 1-bits.

Time Efficiency Consideration :

. It requires 16 iterations for any bit patterns of n. This is very inefficient when n contains a lot of 0-bits. E.g., if n = 16384. It takes 16 iterations to discover that the number of 1-bits in 16384 (2^{14}) is only 1.

A better method to count the number of 1-bits :

In each iteration we hop to the nearest 1-bit and erase it. Repeat these operations until the number becomes 0. The number of iterations is the number of 1-bits.

Example :

n = 0 0 0 0 1 0 0 1 0 0 0 0 1 0 0 0

0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0

0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

It incurs 3 iterations before n becomes 0. The number of 1-bits in the number is 3.

Program

```
/* AC2-2.C */

#include<stdio.h>

int Count_Bits_Better(unsigned n)   /* Count the 1-bit in n */
{
  int count=0;

  while (n>0)
  {
     count++;
     n &= n-1;    /* n = n & (n-1); */
  }

  return count;
}

main()
{
  int number;

  printf("Enter an integer:");
  scanf("%d",&number);
  printf("There are %d 1-bits.\n", Count_Bits_Better(number));

  return 0;
}
```

Screen Output

```
Enter an integer:78
There are 4 1-bits.
```


4. Bit Rotations

. Slightly different from bit shifting.
. Dropped bits are appended to the other end.
  E.g.,
  0001 *0001 0000 0001* left-rotated 4 bits becomes
  0001 0000 0001 0001

  But for shift operation, bits are dropped !!!!
  E.g.,
  0001 *0001 0000 0001* << 4 = *0001 0000 0001* 0000

. No rotate operator in C language.

In the following program, a rotate_l function is written to rotate an integer x to the left by n places. A main function is also included to test the function.

Program

```
/* AC2-3.C */

#include<stdio.h>

int rotate_l(int ,int );

void main()
{
  int x,n,z;

  x=0xfa27;  /* this is any arbitrary number
                                  for demonstration */
  n=4;
  z=rotate_l(x,n);
  printf(" %4x (base 16) rotated left by %d bits
                          becomes %4x (base 16)\n", x,n,z);
}

int rotate_l(int x,int n)
{
  int i,truncate;

  for (i=0;i<n;i++)
  {
     truncate = x & 0x8000;
                          /* drop every bit except the leftmost bit */
     x <<= 1;
                           /* x = x<<1, shift x to the left by 1 bit */
     if (truncate !=0)
       x |= 1;             /* x |= 1, to set the rightmost bit */
  }

  return x;
}
```


truncate = x & 0x8000;

X:	?															
& 0x8000:	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	?	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

x << = 1; will make bit-0 of x to contain 0

If (truncate !=0) x |= 1;   /* x =   x | 1 ;   */

X:																0
| 1:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
																1

Screen Output

fa27 (base 16) rotated left by 4 bits becomes a27f (base 16)

5. Case Study

A twentieth-century date can be written with integers in the form day/month/year. An example is 24/1/95, which represents 24 January 1995. A simple way to store the date is by using a struct definition as follows :

```
struct date
{
   int day;
   int month;
   int year;
};
```

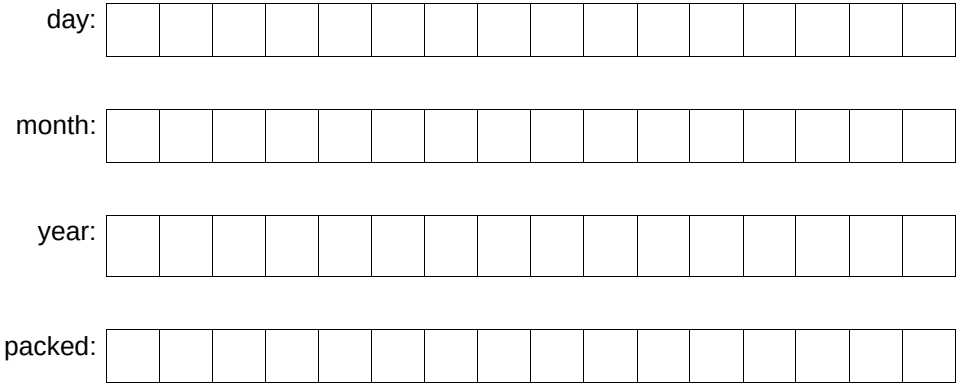
Storage Efficiency Consideration :

- Too much storage overhead; we only require 31 different values for the day, 12 different values for the month and 100 different values for the year, and their values need not, and will never, go up to 32767.
- Need only 5 bits to represent the day, 4 bits to represent the month, and 7 bits to represent the year.

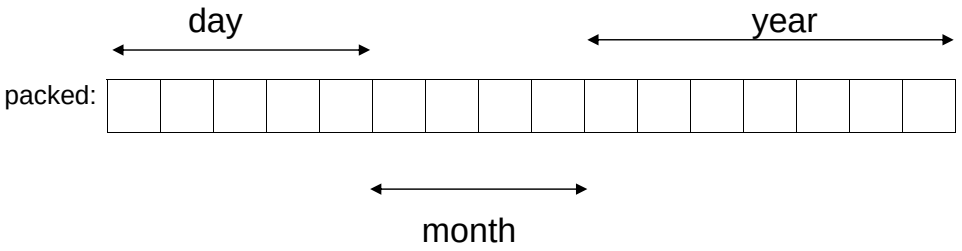
Function pack_date is to perform this task. Function print_bit is to print the packed date. Function unpack_date, is used to test that the date packing is correct.

Program

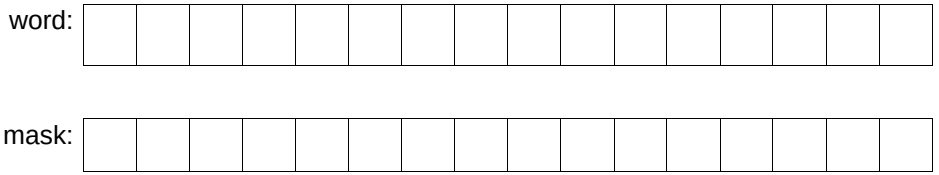
```
/* AC2-4.C */

#include<stdio.h>

unsigned pack_date(int day, int month, int year)
{
   unsigned packed;

   day <<= 11;   /* drop the first 11 bits and
                    shift the contents to the left */
   month <<= 7;  /* drop the first 7 bits and
                    ensure that bits 0 to 6 are zeros. */
   year %= 100;
                    /* keep the last 2 digit of year */
  packed = day | month | year;
  return packed;
}
```



```
void unpack_date(unsigned packed)
{
   int day,month,year;

   day = packed >> 11;
   month = (packed & 0x0780) >> 7;
   year = packed & 0x007f;

   printf("%d/%d/%d\n",day,month,year);
}
```




```
void print_bit(unsigned word)
{
  int n=sizeof(int)*8,i;
  int mask=1<<(n-1);

  for (i=0;i<n;i++)
  {
     if (word & mask)
       printf("1");
     else
       printf("0");
     word <<= 1;
  }
  putchar('\n');
}
```



```
main()
{
  int d,m,y;
  unsigned packed;

  printf("Test Program for date packing.\n");
  printf("Enter a date in the form dd/mm/yyyy : ");
  scanf("%d/%d/%d",&d,&m,&y);
  packed=pack_date(d,m,y);
  printf("The packed bit pattern is : ");
  print_bit(packed);
  printf("Unpacked date is :");
  unpack_date(packed);

  return 0;
}
```

Screen Output

Test Program for date packing.
Enter a date in the form dd/mm/yyyy : *19/11/1994*
The packed bit pattern is : 1001110111011110
Unpacked date is :19/11/94

6. Bits Fields

- Reduced storage space.
- More convenient mechanism likes struct.

But,

- Using bit fields can make a program machine dependent (not portable).

Example :

A sports club with a computer system to record details of members;

. birth date (day, month, year),
. gender (male = 0, female = 1),
. active in a club team (no = 0, yes = 1),
. club dues are paid (no = 0, yes = 1).

This can be done with the following declarations (which would normally form part of a larger record):

```
struct
 {
   unsigned int BirthDay     :5;
                             /* meaning use only 5 bits */
   unsigned int BirthMonth   :4;
                             /* meaning use only 4 bits */
   unsigned int BirthYear    :7;
   unsigned int Female       :1;
   unsigned int Active       :1;
   unsigned int PaidUp       :1;
 } PersonalDetails;
```

The structure members in this case are bit fields. The numbers after the colon following each field represent the

number of bits to be allowed for the field. Thus five bits for the BirthDay allow values in the range 0...31, which is enough for the largest month.

The fields can now be used completely as if they were structure members. E.g.

PersonalDetails.BirthDay = 27;

assigns a value to the Birthday member.

if(PersonalDetails.Active && !PersonalDetails.Female)
:

would select active, male members of the club.

The same effect could be achieved by using bitwise and shift operations to unpack the fields.

Using a bit field causes the compiler to generate the required shifts and saves programmer's effort.

We first write a program to generate the input file members.dat

Data to be Entered

Date of Birth	Gender	Active	PaidUp
13/11/70	1	1	0
20/08/60	0	0	1
01/01/65	0	0	0
31/05/55	0	1	0
10/10/44	0	1	1
25/12/48	0	1	1
22/07/53	1	0	0
02/09/69	0	1	0
13/01/58	0	1	0
05/05/61	1	1	1

Some Essential C Functions and Data Type

size_t is an unsigned integer.

size_t fread(void *Data, size_t ObjSize, size_t NumObjs, FILE *indata)

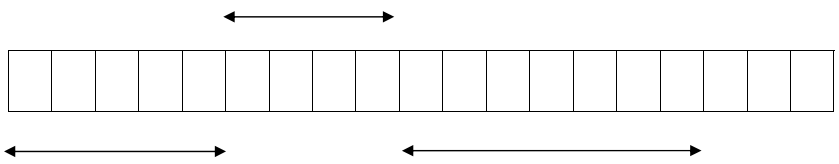
Reads from **indata** to the array pointed to by **Data** up to **NumObjs** objects each of size **ObjSize**. It returns the number of objects (not bytes) actually read, which may be fewer than **NumObjs** if the end of file is met.

size_t fwrite(const void *Data, size_t ObjSize, size_t NumObjs, FILE *outdata)

Writes to **outdata** from the array pointed to by **Data** up to **NumObjs** objects each of size **ObjSize**. It returns the number of objects (not bytes) actually written, which may be fewer than **NumObjs** if a write error occurs.

Program

```
/* AC2-5A.C   - This program creates a binary data file
                named "members.dat" for "AC2-5B.C".     */

#include<stdio.h>

main()
{
  struct
  {
     unsigned int BirthDay     :5;
     unsigned int BirthMonth   :4;
     unsigned int BirthYear    :7;
     unsigned int Female       :1;
     unsigned int Active       :1;
     unsigned int PaidUp       :1;
  } PersonalDetails;

  FILE *outdata;
  char more;
  int data;

  outdata=fopen("members.dat","wb");
  /*  Binary File, not readable  */
```




```
  do {
        printf("Response to the following querying please !!\n");
        printf("BirthDay : ");
        scanf("%d",&data);
        PersonalDetails.BirthDay=data;

        printf("BirthMonth : ");
        scanf("%d",&data);
        PersonalDetails.BirthMonth=data;

        printf("BirthYear : ");
        scanf("%d",&data);
         data = data %100;
        PersonalDetails.BirthYear=data;

        printf("Sex (male=0, female=1) : ");
        scanf("%d",&data);
        PersonalDetails.Female=data;

        printf("Active (no=0, yes=1) : ");
        scanf("%d",&data);
        PersonalDetails.Active=data;
        printf("Paidup : ");

        scanf("%d%*c",&data);
        PersonalDetails.PaidUp=data;
        fwrite(&PersonalDetails,
                    sizeof(PersonalDetails), 1, outdata);
        printf("Any more members?(y/n)");
        scanf("%c",&more);
    }while (more!='n');

  fclose(outdata);

  return 0;
}
```

The following program reads the binary file members.dat where each record is stored as the struct format and print out the birthday of those active male members who have not paid up the club fees. The program also displays the numbers of female and male club members.

Program

```
/* AC2-5B.C */

#include<stdio.h>
#define TRUE 1

main()
{
  struct
  {
    unsigned int BirthDay    :5;
    unsigned int BirthMonth :4;
    unsigned int BirthYear   :7;
    unsigned int Female      :1;
    unsigned int Active      :1;
    unsigned int PaidUp      :1;
  } PersonalDetails;

  FILE *indata;
  int total=0,female=0;

  indata=fopen("a:\\ac2\\members.dat","rb");

  printf ("\nRecords of active male members
                                  who have not paid:\n");
```



```
  do
  {
    fread(&PersonalDetails, sizeof(PersonalDetails), 1, indata);
    if (feof(indata))  break;
    if(PersonalDetails.Active  && !PersonalDetails.Female
          &&  !PersonalDetails.PaidUp)
    {
        printf("\nBirth date : %d/%d/%d\n",
         PersonalDetails.BirthDay, PersonalDetails.BirthMonth,
                                        PersonalDetails.BirthYear);
        printf("Sex : Male.\n");
        printf("Status : Active.\n");
        printf("Paidup : NO.\n");
    }
    total++;
    if (PersonalDetails.Female) female++;
  } while (TRUE);

  printf("\nThere are %d female members and %d male
            members in the club.\n", female, total-female);

  fclose(indata);

  return 0;
}
```

Screen Output

Records of active male members who have not paid:

Birth date : 31/5/55
Sex : Male.
Status : Active.
Paidup : NO.

Birth date : 2/9/69
Sex : Male.
Status : Active.
Paidup : NO.

Birth date : 13/1/58
Sex : Male.
Status : Active.
Paidup : NO.

There are 3 female members and 7 male members in the club.

Order of Bit Fields Storage

- Not defined by the ANSI standard.
- Implementation dependent.
- To reduce the chance of portability problems, avoid fields of more than 16 bits and never make assumptions about the order of storage.
- Fields can be only of type int (signed or unsigned), but as they are only part of a word they do not have addresses, and so cannot be used with the address (&) operator.

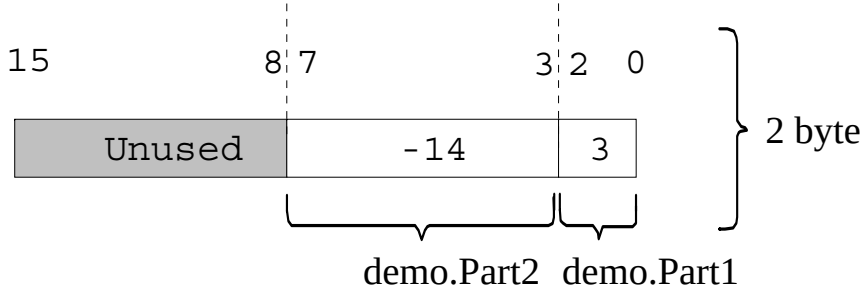
The following two programs illustrate bit alignment (boundary) and the storage requirements.

Program

```
/* AC2-6.C */

#include<stdio.h>

main()
{
  int size;
  struct
  {
     int Part1 : 3;
     int Part2 : 5;
  } demo;

  size=sizeof(demo);
  printf("size of struct is %d bytes.\n",size);
  demo.Part1 = 3;
  demo.Part2 = 18;

  printf("Part1=%d, Part2=%d\n",demo.Part1,demo.Part2);

  return 0;
}
```

Screen Output (with word alignment)

size of struct is 2 bytes.
Part1=3, Part2=-14



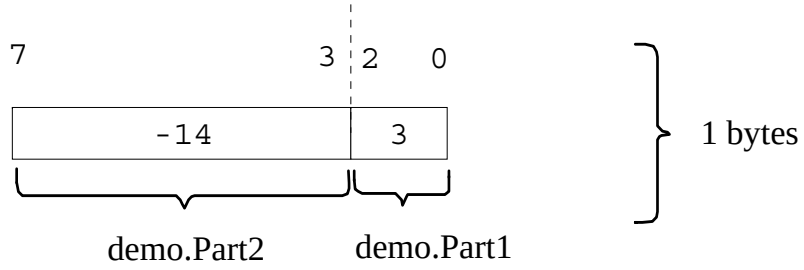
Why –14 and not 18?

Program

```
/* AC2-6.C */

#include<stdio.h>

main()
{
  int size;
  struct
  {
    int Part1 : 3;
    int Part2 : 5;
  } demo;

  size=sizeof(demo);
  printf("size of struct is %d bytes.\n",size);
  demo.Part1 = 3;
  demo.Part2 = 18;

  printf("Part1=%d, Part2=%d\n",demo.Part1,demo.Part2);

  return 0;
}
```

Screen Output (without word alignment)

size of struct is 1 bytes.
Part1=3, Part2=-14

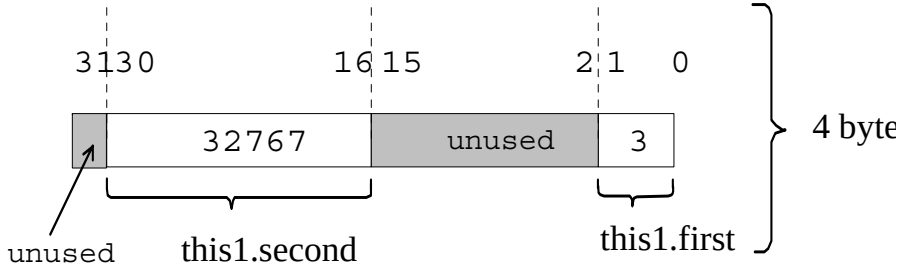


Program

```
/* AC2-7.C */

#include<stdio.h>

main(void)
{
  struct bits
  {
     unsigned first : 2;
     unsigned second : 15;
  };
  struct bits this1;

  this1.first=3;
  this1.second=0x7fff;
  printf("first = %u, second = %u\n", this1.first, this1.second);
  printf("size of struct = %d bytes.",sizeof(this1));

  return 0;
}
```

Screen Output (with word alignment)

first = 3, second = 32767
size of struct = 4 bytes.

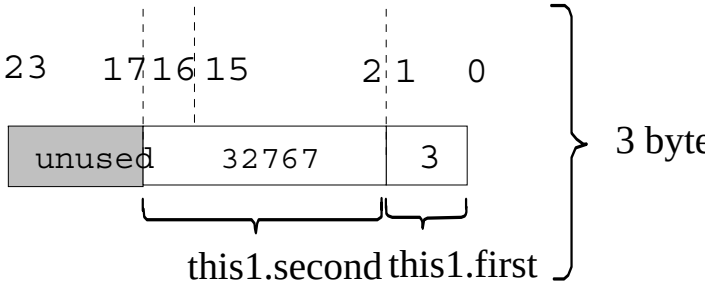


Program

```
/* AC2-7.C */

#include<stdio.h>

main(void)
{
  struct bits
  {
     unsigned first : 2;
     unsigned second : 15;
  };
  struct bits this1;

  this1.first=3;
  this1.second=0x7fff;
  printf("first = %u, second = %u\n", this1.first, this1.second);
  printf("size of struct = %d bytes.",sizeof(this1));

  return 0;
}
```

Screen Output (without word alignment)

first = 3, second = 32767
size of struct = 3 bytes.



Chapter 4

Advanced File Operations

1. Introduction

- **FILE** is defined in **<stdio.h>** and used for input and output.
- Text file is readable, but not binary file.
- Three files are opened when a C program is executing.
  stdin : default input file
  stdout : default output file
  stderr : a file to which error messages are written.
- On microcomputers **stdin** often represents the keyboard, and **stdout** and **stderr** the screen, but this is by no means universal.

2. Opening and Closing Files

- Reference to files is via variables which are always pointers to a structure of type **FILE**.

- A typical declaration might be

  FILE *indata;

  This file can then be opened by

  indata = fopen ("monkey.inf","r");

- Open the file ***monkey.inf*** for reading.

- Generate a structure to hold whatever control information is needed, and leave indata pointing to it.

- File descriptor may be complicated.

  E.g., "a:\\monkey.inf" where a drive name is appended. The second parameter is also a string, "r" in this case meaning 'read'.

How about c:\\cz1102\\prac3\\monkey.inf ?

The options are:

"r"	Open an existing file for reading.
"w"	Open a file for writing. If the file does not exist then it will be created and if it does exist then its contents will be deleted before writing.
"a"	Open an existing file for writing so that new data will be appended to the end of the current file contents.
"r+"	Open a file for reading but allow writing. That is, the file can be both read from and written to. However, errors will be handled as if the file were opened for reading.
"w+"	The same as **"r+"** except that errors will be appropriate to writing.
"a+"	Open a file for appending but allow reading at the same time.

- If a file is to be opened for binary input or output then a 'b' ('b' means binary) should be added to the above strings e.g. **"wb"**, **"r+b"** or **"rb+"**.

- **fopen** will return a pointer value of **NULL** if the file cannot be opened and so this can be used to test for successful completion of the operation.
- The file opened above can be closed with:

  fclose (indata);
- File closing should not be omitted because it is dangerous to do so, particularly for files which are opened for writing.
- Closing a file releases the space used by the **FILE** structure and ensures that, when a file is being written, the file buffer is emptied.
- When a program requests that a character be written to a disk the request is not carried out immediately. Instead the character is placed in an array, called a buffer, which accumulates characters until there are enough to be worth writing. 'Enough', in this sense, usually means the size of a disc sector.
- If the buffer is not emptied then the file may not include the latest data written.
- ***fclose*** ensures that the buffer will be written, i.e., the contents of the buffer are transferred to a storage device.

The following example shows a program to create a sequential file.

Program

```
/* AC4-1.C */
/* NOTE : A file named "clients.dat" will be generated in the current directory */
#include <stdio.h>
main()
{
  int account;
  char name[30];
  float balance;
  FILE *outdata; /*outdata=clients.dat file pointer */

  if ((outdata=fopen("clients.dat","w")) == NULL )
    printf ("File could not be opened\n");
  else
   {
     printf("Enter the account, name, and balance.\n");
     printf("Enter <EOF> character (CTRL+Z in DOS) to end input.\n");
     printf("? ");
     scanf("%d%s%f",&account,name,&balance);

     while (!feof(stdin))
     {
       fprintf(outdata," %d %s %.2f\n", account,name,balance);
       printf("? ");
       scanf("%d%s%f",&account,name,&balance);
     }

     fclose(outdata);   /* close the data file */
   }

   return 0;
}
```

Screen Output

Enter the account, name, and balance.
Enter <EOF> character (CTRL+Z in DOS) to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z

Output File (clients.dat)

100 Jones 24.98
200 Doe 345.67
300 White 0.00
400 Stone -42.16
500 Rich 224.62

The following example illustrates the reading and printing of a sequential file.

Program

```
/* AC4-2.C */
/* NOTE : A file named "clients.dat" must be put in current directory */

#include<stdio.h>

main()
{
  int account;
  char name[30];
  float balance;
  FILE *indata;   /* indata = clients.dat file pointer */

  if ((indata=fopen("clients.dat","r")) == NULL )
    printf ("File could not be opened\n");
  else
  {
    printf ("%-10s%-13s%s\n","Account", "Name","Balance");
                                        /* - means left justified */
    printf ("============================\n");
    fscanf(indata,"%d%s%f",&account, name, &balance);
    while (!feof(indata))
    {
      printf("%-10d%-13s%7.2f\n",account, name, balance);
      fscanf(indata,"%d%s%f",&account, name, &balance);
    }

    fclose(indata);       /* close the data file */
  }

  return 0;
}
```

Screen Output

```
Account    Name          Balance
============================
100        Jones           24.98
200        Doe            345.67
300        White            0.00
400        Stone          -42.16
500        Rich           224.62
```


Reopening a File

freopen ("monkey2.inf", "r", indata);

- Closes the existing file associated with indata and opens the file monkey2.inf.
- Acts in the same way as **fopen** in that it also delivers a pointer to indata and delivers **NULL** if the file cannot be opened.

The following example illustrates the use of freopen.

We want to encode two files. Put the output in the same text file.

Input File-1 (monkey1.inf)

Two sections of commando will attack
Delta-2A at 0200 hour.
Big Bird will give support fire.

Input File-2 (monkey2.inf)

If alpha company is confronted,
call for hawk.

Output File (donkey.ouf)

First encoded message :
Ydl hvxgrlmh lu xlnnzmwl droo zggzxp
Ivogz-2K zg 0200 slfi.
Krt Kriw droo trev hfkklig uriv.

Second endcoded message:
Su zoksz xlnkzmb rh xlmuilmgvw,
xzoo uli szdp.

Program

```
/* AC4-3.C */

#include<stdio.h>

main()
{
  FILE *indata,*outdata;
  char this1;
  int i;

  indata=fopen("a:\\ac4\\monkey1.inf","r");
  outdata=fopen("a:\\ac4\\donkey.ouf","w");
  fprintf(outdata,"First encoded message :\n");

  for (i=0;i<2;i++)
  {
     while (fscanf(indata,"%c",&this1)==1)
     {
       if (this1>='A' && this1<='Z')
          fprintf(outdata,"%c", ((this1-2*'A'+'K')%26)+'A');
       else
          if (this1>='a' && this1<='z') fprintf(outdata,"%c",'z'-(this1-'a'));
        else
          fprintf(outdata,"%c",this1);
     }

     if (i==0)
     {
        fprintf(outdata,"\nSecond endcoded message:\n");
        if (freopen("a:\\ac4\\monkey2.inf","r",indata) == NULL) i=1;
     }
  }
  fclose(indata);
  fclose(outdata);

  return 0;
}
```


Empty a output buffer even before the buffer is full

- ***fflush*** forces the emptying of output buffers.
- Flushing output buffer is essential if a file is open in one of the 'update' modes ("w+", "r+" or "a+") and a change is made from writing to reading.
- Unless fflush is called before finishing a sequence of write operations, the information remaining in the output buffer will not be written at the correct point in the file.

This program illustrates the use of **fflush** function.

Program

```
/* AC4-4.C */

#include<stdio.h>

main()
{
  int i;
  float result,mark,sum=0;
  FILE *outdata;

  outdata=fopen("test.dat","w+");
  for (i=0;i<5;i++)
  {
    printf("Enter test result>");
    scanf("%f%*c",&result);
    fprintf(outdata,"%f\n",result);
  }
  fflush(outdata);     /* transfer data from buffer to disk */

  rewind(outdata);
  for (i=0;i<5;i++)
  {
    fscanf(outdata,"%f",&mark);
    sum += mark;
  }

  fclose(outdata);
  printf("Average = %f\n",sum/5);

  return 0;
}
```


Screen Output

```
Enter test result>100.000000
Enter test result>45.000000
Enter test result>83.000000
Enter test result>97.000000
Enter test result>60.000000

Average = 77.000000
```

3. File Positioning

void rewind (FILE *Stream)

Positions **Stream** so that the next read or write operation will be at the beginning of the file. If this is not possible then there will be no effect.

long int ftell (FILE *Stream)

Returns the current positions within **Stream**, or -1L if an error occurs.

int fseek (FILE *Stream, long int Offset, int Origin)

Sets the file position within Stream so that subsequent reading or writing will occur from there. the position is in the form of an **Offset** relative to an **Origin**. The possible values of **Origin** are **SEEK_SET**, **SEEK_CUR** and **SEEK_END**. If an error occurs then a non-zero value is returned; a return value of zero indicates success.

Origin	Measure offset from
SEEK_SET	Beginning of file
SEEK_CUR	Current position
SEEK_END	End of file

Program

```
/* AC4-5.C */

#include<stdio.h>
#include<stdlib.h>

#define MAX 20

main()
{
  FILE *indata;
  char words[MAX];

  if ((indata=fopen("words.ouf","a+")) == NULL)
  {
    fprintf(stderr,"Can't open \"words\" file.\n");
    exit(1);
  }

  puts("Enter words to add to the file; type the <Enter> key \n");
  puts("at the beginning of a line to terminate.");

  while (gets(words)!=NULL && words[0]!='\0')
    fprintf(indata,"%s\n",words);

  puts("File contents:");
  rewind(indata);    /* go back to the beginning of the file */

  while (fscanf(indata,"%s",words)==1)
    puts(words);

  fclose(indata);

  return 0;
}
```

Screen Output

C>*ac4-5*
Enter words to add to the file; type the <Enter> key
at the beginning of a line to terminate.
See the canoes

File contents:
See
the
canoes

C>*ac4-5*
Enter words to add to the file; type the <Enter> key
at the beginning of a line to terminate.
on the
sea

File contents:
See
the
canoes
on
the
sea

Output File (words.ouf)

See the canoes
on the
sea

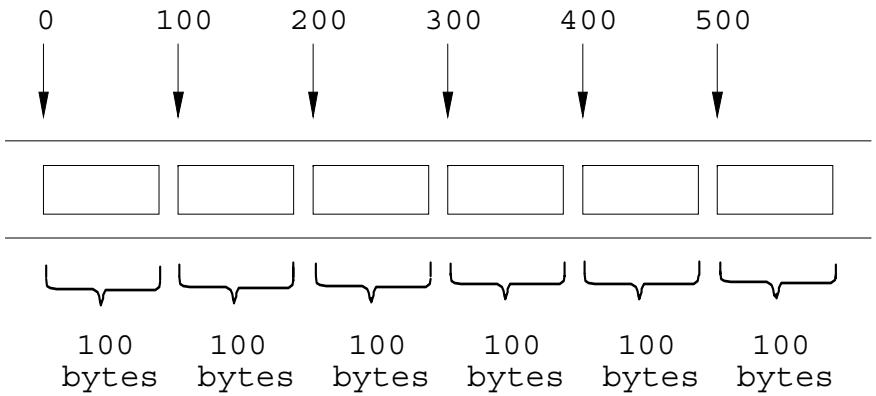
4. Random Access Files

- Records in a file created with the formatted output function **fprintf** are not necessarily of the same length.

- Individual records of a randomly accessed file are normally fixed in length and may be accessed directly (and thus quickly) without searching through other records.

- Randomly accessed files are appropriate for airline reservation systems, banking systems, point-of-sale systems, and other kinds of transaction processing systems that require rapid access to specific data.

- Because every record in a randomly accessed file normally has the same length, the exact location of a record relative to the beginning of the file can be calculated as a function of the record key. We will soon see how this facilitates immediate access to specific records, even in large files.



- Data can be inserted in a randomly accessed file without destroying other data in the file.
- Data stored previously can also be updated or deleted without rewriting the entire file.

Essential Instructions

size_t is an unsigned integer in Turbo C.

size_t fread(void *Data, size_t ObjSize, size_t NumObjs, FILE *indata)

Reads from **indata** to the array pointed to by **Data** up to **NumObjs** objects each of size **ObjSize**. It returns the number of objects (not bytes) actually read, which may be fewer than **NumObjs** if the end of file is met.

size_t fwrite(const void *Data, size_t ObjSize, size_t NumObjs, FILE *outdata)

Writes to **outdata** from the array pointed to by **Data** up to **NumObjs** objects each of size **ObjSize**. It returns the number of objects (not bytes) actually written, which may be fewer than **NumObjs** if a write error occurs.

File processing programs rarely write a single field to a file. Normally, they write one struct at a time.

Example:

We are going to create a credit processing system capable of storing up to 100 fixed-length records. Each record should consist of an account number that will be used as the record key, a last name, a first name, and a balance. The resulting program should be able to update an account, insert a new account record, delete an account, and list all the account records in a formatted text file for printing. Randomly accessed file is used.

Creating a Random Access File.

Program

```
/* AC4-6.C */
/* NOTE : A data file named "credit.dat" will be created for program
        "AC4-7.C" and "AC4-8.C"
*/

#include <stdio.h>

struct clientData
{
  int acctNum;
  char lastName[15];
  char firstName[10];
  float balance;
};

main()
{
  int i;
  struct clientData blankClient = { 0, "", "", 0.0 };
  FILE *outdata;

  if ((outdata=fopen("credit.dat","w")) == NULL)
    printf("File could not be opened.\n");
  else
    {
      for (i=1;i<=100;i++)      /* write 100 blank records to the file */
         fwrite(&blankClient, sizeof(struct clientData), 1, outdata);

      fclose(outdata);
    }

  return 0;
}
```

Output File (credit.dat, binary format and not readable)

- The program initializes all 100 records of the file **"credit.dat"** with empty structs using function **fwrite**.

- Each empty struct contains 0 for the account number, **NULL** (represented by empty quotation marks) for the last name, **NULL** for the first name, and 0.0 for the balance.

- The file is initialized in this manner to create space on the disk in which the file will be stored, and to make it possible to determine if a record contains data.

fwrite

fwrite (&blankClient, sizeof(struct clientData), 1, outdata);

Writing Data Randomly to a Random Access File

- The next program writes data to the file "credit.dat".

- It uses the combination of fseek and fwrite to store data at specific locations in the file. Function **fseek** sets the file position pointer to a specific position in the file, then **fwrite** writes the data.

Program

```
/* AC4-7.C */
/* Writing to a random access file */
#include<stdio.h>
struct clientData
{
   int acctNum;
   char lastName[15];
   char firstName[10];
   float balance;
};
main()
{
  struct clientData client;
  FILE *outdata;
  if ((outdata=fopen("credit.dat","r+")) == NULL)
    printf("File could not be opened.\n");
  else
    {
      printf("Enter account number (1 to 100, 0 to end input)\n? ");
      scanf("%d",&client.acctNum);
      while (client.acctNum != 0)
      {
        printf("Enter lastname,firstname,balance\n?");
        scanf("%s%s%f",client.lastName,client.firstName, &client.balance);
        fseek(outdata,(client.acctNum - 1)* sizeof(struct clientData),
                                                    SEEK_SET);
        fwrite(&client, sizeof(struct clientData), 1, outdata);
        printf("Enter account number\n? ");
        scanf("%d",&client.acctNum);
      }
    }
  fclose(outdata);
  return 0;
}
```


fseek

fseek(outdata, (client.accNum - 1) * sizeof(struct clientData), SEEK_SET);

Screen Output

```
Enter account number (1 to 100, 0 to end input)
? 30
Enter lastname, firstname, balance
Kassim Abula 75.3
Enter account number
? 27
Enter lastname, firstname, balance
Lily Tay 42.5
Enter account number
? 56
Enter lastname, firstname, balance
Angeelo Ali 70.89
Enter account number
? 34
Enter lastname, firstname, balance
Keng Heng  0.00
Enter account number
? 10
Enter lastname, firstname, balance
Milly Ken 203.41
Enter account number
? 0
```

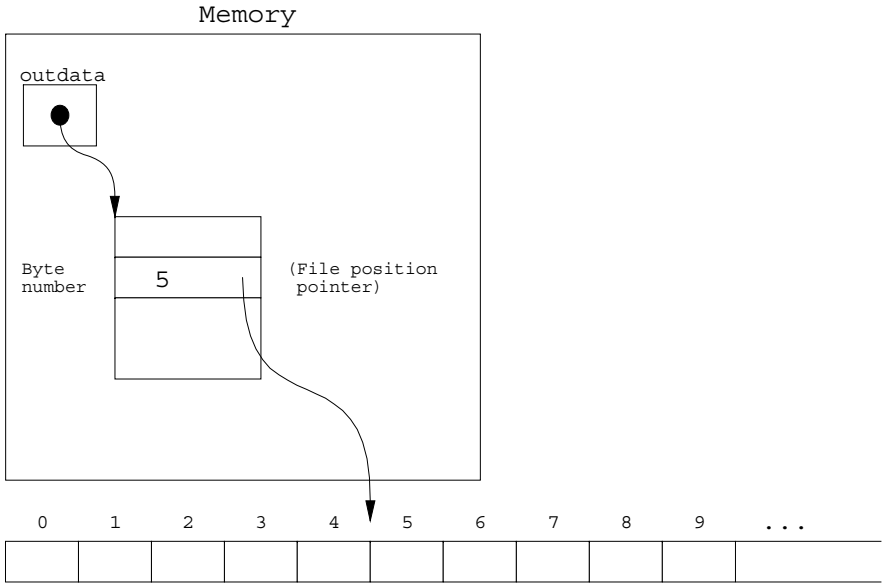
Other Use of fseek

int fseek(FILE *stream, long int offset, int whence);

offset : is the number of bytes from location whence

whence : SEEK_SET
(origin) SEEK_CUR
SEEK_END

These three symbolic constants are defined in the stdio.h header file.



The file position pointer indicating an offset of 5 bytes from the beginning of the file.

Reading Data Randomly from a Random Access File

- Use of fseek and fread, or fseek and fwrite to read/write records from/to a file randomly.
- Position the file pointer to the desired record we want to read or write, then we perform the operation on the record.

Program

```
/* AC4-8.C */

#include<stdio.h>

struct clientData
{
  int acctNum;
  char lastName[15];
  char firstName[10];
  float balance;
};

main()
{
  struct clientData client;
  FILE *indata;
  int option,accountNum;
```

```
  if ((indata=fopen("credit.dat","r+")) == NULL)
    printf("File could not be opened.\n");
  else
  {
    printf("1. Read a record\n");
    printf("2. Change a record\n");
    printf("Enter your choice >");
    scanf("%d%*c",&option);

    printf("Enter account number >");
    scanf("%d%*c",&accountNum);
    fseek(indata, (accountNum-1)*sizeof(struct clientData), SEEK_SET);

    if (option==1)
     {
       fread(&client, sizeof(struct clientData), 1, indata);
       if (client.acctNum==0)
          printf("Account %d has no information.\n", accountNum);
       else
          printf("%-6d %-10s %-11s %10.2f\n",
            client.acctNum, client.lastName, client.firstName, client.balance);
     }
    else
     {
       printf("Enter lastname, first name, balance\n");
       scanf("%s%s%f",client.lastName, client.firstName, &client.balance);
       client.acctNum=accountNum;
       fwrite(&client, sizeof(struct clientData), 1, indata);
     }
  }

  fclose(indata);
  return 0;
}
```


Screen Output

```
1. Read a record
2. Change a record
Enter your choice >1
Enter account number >27
27     Lily       Tay            42.50

1. Read a record
2. Change a record
Enter your choice >2
Enter account number >34
Enter lastname, first name, balance
Keng Heng 100.00

1. Read a record
2. Change a record
Enter your choice >1
Enter account number >80
Account 80 has no information.
```

Chapter 5

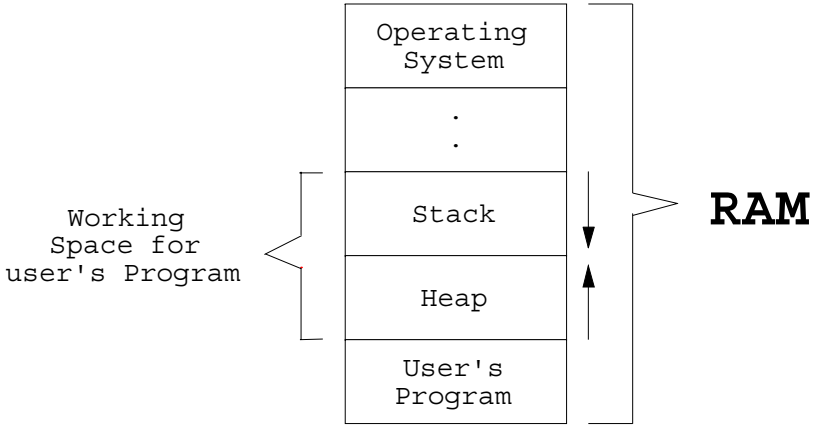
Dynamic Memory Allocation

1 Introduction

- Some memory storage requirement cannot be determined at compilation time, such as the keyboard input and I/O requests.

- The storage is allocated dynamically at runtime. A program area called the heap is used to allocate memory dynamically.

- The heap is kept separate from the stack. It's possible, however, for the heap and stack to share the same memory segment.



A Simplified Semantic View of Memory

- A program that uses a large amount of memory for the stack may have a small amount of memory available for the heap and vice versa.

- You need to ensure that the storage is successfully allocated to store the data and to have some exception handling logic for the unsuccessful storage allocation.

- The heap is controlled by a heap manager, which allocates and deallocates (or returns) memory. User programs interface to the heap via C library calls.

2 C Functions for Dynamic Memory Allocation

Bytes of memory are allocated in user-defined "chunks" (sometimes called objects) with

- **malloc()**
- **calloc()**
- **realloc()**
- **free()** relinquishes memory

Routine	Meaning
char *malloc(size) unsigned size;	allocate storage for size bytes
char *calloc(n, size) unsigned n, size;	allocate and zero storage for n items of size bytes
char *realloc(pheap, newsize) char *pheap; unsigned newsize;	reallocate storage for old heap pointer pheap for newsize bytes
void free(pheap) char *pheap;	free storage for heap pointer pheap

- ***sizeof()*** determines the number of bytes in the structure and makes the code portable.
- ***malloc()*** returns a heap address, and the program casts its return value from a pointer to a char (a byte pointer) to a pointer to a structure (structure block). You should always check the return value from malloc() and the other library calls before you use the heap address. A NULL (defined in stdio.h) indicates that the heap manager was unable to allocate storage.

Program

```
/* AC3-1.C - allocate block on heap */
#include<stdio.h>
#include<stdlib.h>

struct block
{
  int header;
  char data[1024];
};

main()
{
  struct block *p;

  p=(struct block*) malloc (sizeof(struct block));

  if (p == NULL)
   {
     printf("malloc can't allocate heap space.\n");
     exit (1);
   }
  else
   {
     printf("Memory allocation successful!\n");
     free(p);       /* deallocate the memory */
   }

   return 0;
}
```

Screen Output

```
Memory allocation successful!
```


calloc()

- calloc() is similar to malloc(), but the routine fills heap memory with zeros. calloc(), therefore, runs slightly slower than malloc().
- calloc() takes two arguments, which are the number of objects to be allocated, and the size of each object.

E.g.,

```
p2=(struct block*) calloc(MAXENTRIES,
                          sizeof(struct block));
```

- calloc() also returns a heap address.

Program

```
/* AC3-2.C - allocate MAXENTRIES blocks on heap */
#include<stdio.h>
#include<stdlib.h>
#define MAXENTRIES 5

struct block
{
  int header;
  char data[1024];
};

main()
{
  struct block *p1,*p2;

  p1=(struct block*) malloc(MAXENTRIES * sizeof(struct block));
  if (p1 == NULL)
    {
      printf("malloc can't allocate heap space.\n");
      exit (1);
    }
  else
    {
      printf("Memory allocation for 'p1' is successful!\n");
      free(p1); /* return memory to the system */
    }
```



```
  p2=(struct block*) calloc(MAXENTRIES,  sizeof(struct block));

  if (p2 == NULL)
    {
      printf("calloc can't allocate heap space.\n");
      exit (1);
    }
  else
    {
      printf("Memory allocation for 'p2' is successful!\n");
      free(p2); /* return memory to the system */
    }

    return 0;
}
```

Screen Output

```
Memory allocation for 'p1' is successful!
Memory allocation for 'p2' is successful!
```

- After the call to malloc(), p1 points to the first of five consecutive structures (of type block) in memory.
- Similarly, p2 points to the first of five structures after the call to calloc(). The storage that p2 points to is zero-filled. Note that we pass two arguments to calloc(), but only one to malloc().
- Each call allocates the same amount of heap memory.

realloc()

- **realloc()** allows you to change the size of any object on the heap.
- The routine's first argument is a pointer to a heap address. Presumably, this pointer is initialized from a previous call to the heap manager. The second argument is the number of bytes (including the existing) to be reallocated.

  E.g.,

```
if ((num=(int*) realloc(num,size*(j+1)))==NULL)
{
  printf("realloc fails.\n");
  exit(1);
}
```

- **realloc()** can increase or decrease an object's size in heap memory.

- **realloc()** also preserves data in memory. If the storage space is being increased and there's not enough contiguous space on the heap, realloc() returns a **different address** than the one that you pass as its first argument. This means realloc() may have to move data; hence, its execution time varies.

- Occasionally, realloc() returns the *same* heap address. So, you have to maintain a table of heap addresses. When you allocate memory from the heap manager using any of the library calls, you should always update your table with the new pointer, even though it may not have changed. Programs that don't do this are not reliable.

- The heap manager frees heap memory with the C library call free(). You call it with a heap address returned from a previous call to malloc(), calloc(), or realloc(). The following statements free a structure called block from the heap:

```
struct block *p;

p = (struct block *) malloc (sizeof(struct block));

if (p == (struct block *) NULL)
{
  printf("malloc can't allocate heap space\n");
  exit(1);
}


/* processing on the block */
            .
            .
free(p);    /* free the structure */
            .
            .
```

You must pass a pointer to the start of some previously allocated space to free(). Note that free() doesn't return any thing.

4 Building an Array on the Fly

Suppose you are told to write a C program to perform a set of statistical analysis on the fly, how do you use array in a program without knowing the sample size ?

The following program demonstrates the way to build an array of ***arbitrary size***.

Program

```
/* AC3-3.c */

#include<stdio.h>
#include<stdlib.h>

main()
{
  int count,size,i,j;
  int *num,sum=0;
  float average;

  printf("Enter as many integer values as you want! \n");
  printf("I will build an array on the fly with them
                                and compute the average\n");
  printf("and the number of occurrences of the
                               numbers which are less than\n");
  printf("the average.\n");
  printf("Note : Any non-number means you are done.\n");

  size = sizeof(int);
  if ((num=(int*) malloc (size))==NULL)
  {
    printf("malloc fails.\n");
    exit(1);
  }

  j=0;
```



```
  while (scanf("%d%*c",&num[j])==1)
  {
    sum+=num[j];
    j++;

    /* enlarge the array */
    if ((num=(int*) realloc(num,size*(j+1)))==NULL)
    {
      printf("realloc fails.\n");
      exit(1);
    }
  }

  average=(float)sum/j;
  count=0;
  for (i=0;i<j;i++) if (num[i]<average) count++;

  printf("Average = %f\n",average);
  printf("No. of occurrences below average = %d\n", count);
  return 0;
}
```

Screen Output

```
Enter as many integer values as you want!
I will build an array on the fly with them and compute the average
and the number of occurrences of the numbers which are less than
the average.
Note : Any non-number means you are done.
2800
3500
2000
1800
2200
3800
1800
4000
x

Average = 2737.500000
No. of occurrences below average = 4
```

- The previous program is flexible as the size of the array can vary at runtime.
- However, the program is not efficient as the realloc function is invoked in each iteration.

The next program shows an example of dynamic memory allocation, but of a larger grain size. The program allocates a *contiguous 10 bytes* on the heap whenever the realloc function is called. Finally, the program *trims away* the storage which was allocated earlier but *not occupied*.

Program

```
/* AC3-4.C */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

main()
{
  char *first,*current;
  char this1;
  int  buffersize=10,increment=10,count=0;

  printf("\n Enter a string of any length and ");
  printf("\n press 'enter' key when you are done :");

  if ((first=current=malloc(buffersize))==NULL)
  {
    printf("malloc fails.\n");
    exit(1);
  }
```



```
  scanf("%c",&this1);
  while (this1 != '\n')
  {
    count++;
    *current=this1;

    if (count%increment==0)
      {
        buffersize+=increment;
        if ((first = current = realloc(first,buffersize))==NULL)
        {
            printf("realloc fails.\n");
            exit(1);
        }
        current += count;
      }
    else
      current++;

    scanf("%c",&this1);
  }
```

```
  *current='\0';
  count++;
  first=realloc(first,count);
  printf(" input string = %s\nlength = %d\n", first,strlen(first));

  return 0;
}
```

Screen Output

```
Enter a string of any length and
press 'enter' key when you are done :C is so powerful that a lot of
scientists and engineers have switched from other languages to it.

input string = C is so powerful that a lot of
scientists and engineers have switched from other languages to it.
length = 96
```

5 Arrays of Pointers

- C lets you create arrays of any type of elements.
- You can even create an array whose elements are pointers.

For example, to create an array of 10 pointers, in which each item is a pointer to a float, simply declare the following :

```
float  *array_name[10];
```

- The * preceding the array name in this declaration tells the compiler that the array is an *array of pointers*; therefore, *each element holds an address (pronounced as **where**)*.

- The float signifies that all pointers will point to float variables.

You can use this technique for speeding up some sorting programs. Suppose we are going to sort student records in the ascending order of matrix number where each record is defined as follows :

```
struct student
{
  char  matrix[10];
  char  dob[10];
  char  sex;
  char  subject[28][6];
};
```

Remember that for sorting data, we have to interchange their contents if they are not in order:

```
int temp, i, j;

if (i > j)
{
  temp = i;
  i = j;
  j = temp;
}
```

It incurs 3 assignment instructions for each interchange. Now in sorting our student records, each interchange involves

3 * (1+1+1+28) = 93 assignment instructions.

This is too expensive in terms of CPU time !!!

In the next program, we will use selection sort to sort an array of pointers, pointing to student records. Take note that if the matriculation numbers of two records are not in order, we will only interchange the pointers pointing to the records. The contents of the records remain unchanged.

Input File (students.inf)

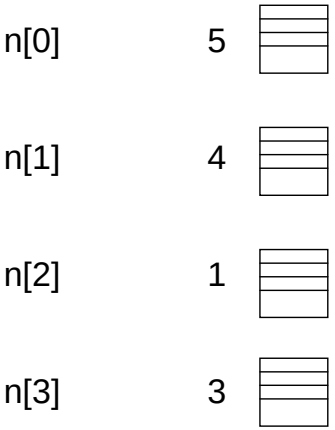
```
942343D02 01/06/75 M CP111 CM101 GM101 . . .
946785U03 03/02/74 F CP112 PC101 MA101 . . .
945786V04 16/12/73 M PC111 CZ101 MA101
947894P01 23/11/76 F CP111 BA123 GM101
945676Z02 23/09/75 M CP112 DB101 GM101
947983X02 11/03/73 M CP111 CM101 GM101
948797U03 23/12/74 F CP112 PC101 MA101
943664V04 13/08/73 M BA111 CZ101 MA101
944865P01 20/01/74 M CP111 BA123 GM102
944564Z02 30/09/75 M CP101 PC101 GM101
```

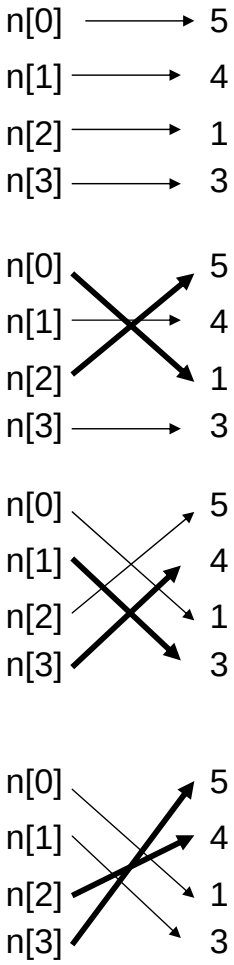
Selection Sort

Unsorted Array 1st iteration 2nd iteration 3rd iteration

5
4
1
3

Selection Sort Using Pointers:





strcmp

Function Compares one string to another

Syntax
include <string.h>
int strcmp (const char **s1*, const char **s2*);

Remarks strcmp performs an unsigned comparison of *s1* and *s2*, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the string is reached.

Returns value

strcmp returns a value that is
< 0 if *s1* is less than *s2*
== 0 if *s1* is the same as *s2*
>0 if *s1* is greater than *s2*

Program

```
/* AC3-5.C */

#include<stdio.h>
#include<stdlib.h>
#define MAXSIZE 40

struct student
{
   char matrix[10];
   char dob[10];
   char sex;
   char subject[3][6];  /* To make life easy, I have declared a smaller
         two dimensional array of characters which is slightly different
         from the one mentioned earlier. But the actual sorting
         strategy used in this program can be applied to any amount
         of subjects. */
  } *st[MAXSIZE];

int GetRecord(void);
void SortRecord(int);

main()
{
  int no_of_st,i;

  no_of_st=GetRecord();
  SortRecord(no_of_st);
  for (i=0;i<no_of_st;i++)
  {
    printf("Matrix No. : %s.\t\tDate of Birth : %s.  \tSex : %c.\n",
              st[i]->matrix,st[i]->dob,st[i]->sex);
    printf("Subjects : %10s%10s%10s.\n\n",  st[i]->subject[0],
                          st[i]->subject[1], st[i]->subject[2]);
  }

  return 0;
}
```



```
GetRecord()
{
  int count=0,i;
  FILE *indata;

  indata=fopen("students.inf","r");
  do
  {
    st[count]=(struct student*)malloc(sizeof (struct student));
    fscanf(indata,"%s%s%*c%c%s%s%s", st[count]->matrix,
           st[count]->dob, &st[count]->sex,st[count]->subject[0],
           st[count]->subject[1], st[count]->subject[2] );

    count++;
  } while (!feof(indata));

  fclose(indata);

  return count;
}


void SortRecord(int n)         /*  Selection sort s used  */
{
  int i,j,min;
  struct student *temp;

  for (i=0;i<n;i++)
  {
    min=i;
    for (j=i+1;j<n;j++)  /* find the student with
                                        the smallest matrix No. */
     if (strcmp(st[j]->matrix,st[min]->matrix)<0)  min=j;

     if (min!=i)
     {
       temp=st[min];      /* only 3 interchanges */
       st[min]=st[i];
       st[i]=temp;
     }
  }
}
```

st[0]
st[1]
st[2]
st[3]
:
:
st[39]

matric[10]	dob	sex	sub[0]	sub[1]	sub[2]
matric[10]	dob	sex	sub[0]	sub[1]	sub[2]
matric[10]	dob	sex	sub[0]	sub[1]	sub[2]
matric[10]	dob	sex	sub[0]	sub[1]	sub[2]

:
st[i]
:
:
:
:
st[n-1]

945786V04					
943664V04					

Screen Output

Matrix No. : 942343D02. Date of Birth : 01/06/75. Sex : M.
Subjects : CP111 CM101 GM101.

Matrix No. : 943664V04. Date of Birth : 13/08/73. Sex : M.
Subjects : BA111 CZ101 MA101.

Matrix No. : 944564Z02. Date of Birth : 30/09/75. Sex : M.
Subjects : CP101 PC101 GM101.

Matrix No. : 944865P01. Date of Birth : 20/01/74. Sex : M.
Subjects : CP111 BA123 GM102.

Matrix No. : 945676Z02. Date of Birth : 23/09/75. Sex : M.
Subjects : CP112 DB101 GM101.

Matrix No. : 945786V04. Date of Birth : 16/12/73. Sex : M.
Subjects : PC111 CZ101 MA101.

Matrix No. : 946785U03. Date of Birth : 03/02/74. Sex : F.
Subjects : CP112 PC101 MA101.

Matrix No. : 947894P01. Date of Birth : 23/11/76. Sex : F.
Subjects : CP111 BA123 GM101.

Matrix No. : 947983X02. Date of Birth : 11/03/73. Sex : M.
Subjects : CP111 CM101 GM101.

Matrix No. : 948797U03. Date of Birth : 23/12/74. Sex : F.
Subjects : CP112 PC101 MA101.