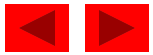
Files in C



© 2000 Prentice Hall, Inc.
All rights reserved.

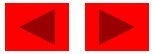
What is a file?

- A named collection of data, stored in secondary storage (typically).
- Typical operations on files:
  - Open
  - Read
  - Write
  - Close
- How is a file stored?
  - Stored as sequence of bytes, logically contiguous (may not be physically contiguous on disk).

l, Inc.
All rights reserved.

- The last byte of a file contains the end-of-file character (EOF), with ASCII code 1A (hex).
- While reading a text file, the EOF character can be checked to know the end.

- **Two kinds of files:**
  - Text :: contains ASCII codes only
  - Binary :: can contain non-ASCII characters
    - Image, audio, video, executable, etc.
    - To check the end of file here, the *file size* value (also stored on disk) needs to be checked.



© 2000 Prentice Hall, Inc.
All rights reserved.

File handling in C

- In C we use `FILE *` to represent a pointer to a file.
- `fopen` is used to open a file. It returns the special value `NULL` to indicate that it is unable to open the file.

```
FILE *fptr;
char filename[]= "file2.dat";
fptr = fopen (filename,"w");
if (fptr == NULL) {
  printf ("ERROR IN FILE CREATION");
     /* DO SOMETHING */
}
```

© 2000 Prentice Hall, Inc.
All rights reserved.

Modes for opening files

- The second argument of `fopen` is the *mode* in which we open the file. There are three modes.

  "r" opens a file for reading.

  "w" creates a file for writing, and writes over all previous contents (deletes the file so be careful!).

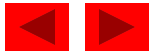
  "a" opens a file for appending – writing on the end of the file.

- We can add a "b" character to indicate that the file is a *binary* file.
  - "rb", "wb" or "ab"

```
fptr = fopen ("xyz.jpg", "rb");
```

- Table of file open modes:

Mode	Description
r	Open a file for reading.
w	Create a file for writing. If the file already exists, discard the current contents.
a	Append; open or create a file for writing at end of file.
r+	Open a file for update (reading and writing).
w+	Create a file for update. If the file already exists, discard the current contents.
a+	Append; open or create a file for update; writing is done at the end of the file.



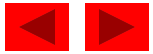
© 2000 Prentice Hall, Inc.
All rights reserved.

Writing a File :

```
#include<stdio.h>
int main(){
        FILE *fp;
        int ch;
        fp=fopen("abc.txt","w");
        printf("Enter text! ^Z");
        while((ch=getchar())!=EOF){
                fputc(ch,fp);
        }
        printf("\n File Written Successfully!");
        fclose(fp);
}
```

Reading from a File :

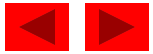
```
#include<stdio.h>
int main(){
        FILE *fp;
        int ch;
        fp=fopen("abc.txt","r");
        printf("Enter text! ^Z");
        while((ch=fgetc(fp))!=EOF){
                printf("%c",ch);
        }
        fclose(fp);
}
```



© 2000 Prentice Hall, Inc.
All rights reserved.

Reading and writing a file

```
 int main(){
FILE *fp1,*fp2;
int ch;
fp1=fopen("xyz.txt","w");
if(fp1==NULL){
                printf("Couldn't able to write!");
                return;
}
fp2=fopen("abc.txt","r");
if(fp2==NULL){
                printf("Couldn't able to read!");
                return;
}
while((ch=fgetc(fp2))!=EOF){
fputc(ch,fp1);
}
fclose(fp1);
fclose(fp2);
}
```

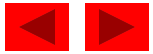


© 2000 Prentice Hall, Inc.
All rights reserved.

Closing a file

- We can close a file simply using `fclose()` and the file pointer.

```
FILE *fptr;
char filename[]= "myfile.dat";
fptr = fopen (filename,"w");
if (fptr == NULL) {
    printf ("Cannot open file to write!\n");
    exit(-1);
}
fprintf (fptr,"Hello World of filing!\n");
fclose (fptr);
```



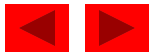
© 2000 Prentice Hall, Inc.
All rights reserved.

Reading lines from a file using fgets( )

We can read a string using `fgets()`.

```
FILE *fptr;
char line [1000];
/* Open file and check it is open */
while (fgets(line,1000,fptr) != NULL)
{
   printf ("Read line %s\n",line);
}
```

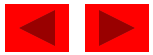
`fgets()` takes 3 arguments – a string, maximum number of characters to read, and a file pointer. It returns **NULL** if there is an error (such as **EOF**).



© 2000 Prentice Hall, Inc.
All rights reserved.

fgets()

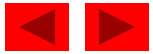
```
#include<stdio.h>
#include<string.h>
int main(){
        FILE *fp1;
        char str[80];
        fp1=fopen("test","r");
        if(fp1==NULL){
                printf("Couldn't able to write!");
                return;
        }
        printf("\n Enter the text..end with ^Z");

        while(fgets(str,80,fp1)!=NULL){
                puts(str);
        }
        fclose(fp1);
}
```



© 2000 Prentice Hall, Inc.
All rights reserved.

fputs()

```
#include<stdio.h>
#include<string.h>
int main(){
        FILE *fp1;
        char str[80];
        fp1=fopen("test","w");
        if(fp1==NULL){
                printf("Couldn't able to write!");
                return;
        }
        printf("\n Enter the text..end with ^Z");

        while(gets(str)!=NULL){
                strcat(str,"\n");
                fputs(str,fp1);
        }
        fclose(fp1);
}
```

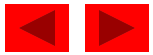


© 2000 Prentice Hall, Inc.
All rights reserved.

Writing to a file using fprintf( )

- `fprintf()` works just like `printf()` and `sprintf()` except that its first argument is a file pointer.

```
FILE *fptr;
Fptr = fopen ("file.dat","w");
/* Check it's open */

fprintf (fptr, "Hello World!\n");
fprintf (fptr, "%d %d", a, b);
```

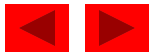


© 2000 Prentice Hall, Inc.
All rights reserved.

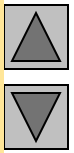
Reading Data Using fscanf( )

- We also read data from a file using `fscanf()`.

```
FILE *fptr;
Fptr = fopen ("input.dat", "r");
/* Check it's open */
if (fptr == NULL)
   {
     printf("Error in opening file \n");
   }
fscanf (fptr, "%d %d",&x, &y);
```



© 2000 Prentice Hall, Inc.
All rights reserved.



Outline

```
1  // ss -> 09-03-2018
2     Create a sequential file */
3  #include <stdio.h>
4
5  int main()
6  {
7     int account;
8     char name[ 30 ];
9     double balance;
10    FILE *cfPtr;   /* cfPtr = clients.dat file pointer */
11
12    if ( ( cfPtr = fopen( "clients.dat", "w" ) ) == NULL )
13       printf( "File could not be opened\n" );
14    else {
15       printf( "Enter the account, name, and balance.\n" );
16       printf( "Enter EOF to end input.\n" );
17
18       scanf( "%d%s%lf", &account, name, &balance );
19
20       while ( !feof( stdin ) ) {    // feof() - to detect EOF
21          fprintf( cfPtr, "%d %s %.2f\n",
22                   account, name, balance );
23
24          scanf( "%d%s%lf", &account, name, &balance );
25       }
26
27       fclose( cfPtr );
28    }
29
30    return 0;
31 }
```

1. Initialize variables and `FILE` pointer

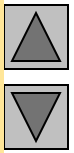
1.1 Link the pointer to a file

2. Input data

2.1 Write to file (`fprintf`)

3. Close file

© 2000 Prentice Hall, Inc.
All rights reserved.



Outline

```
1  /* ss
2     Reading and printing a sequential file */
3  #include <stdio.h>
4
5  int main()
6  {
7     int account;
8     char name[ 30 ];
9     double balance;
10    FILE *cfPtr;   /* cfPtr = clients.dat file pointer */
11
12    if ( ( cfPtr = fopen( "clients.dat", "r" ) ) == NULL )
13       printf( "File could not be opened\n" );
14    else {
15       printf( "%-10s%-13s%s\n", "Account", "Name", "Balance" );
16       fscanf( cfPtr, "%d%s%lf", &account, name, &balance );
17
18       while ( !feof( cfPtr ) ) {
19          printf( "%-10d%-13s%7.2f\n", account, name, balance );
20          fscanf( cfPtr, "%d%s%lf", &account, name, &balance );
21       }
22
23       fclose( cfPtr );
24    }
25
26    return 0;
27 }
```

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

1. Initialize variables

1.1 Link pointer to file

2. Read data (`fscanf`)

2.1 Print

3. Close file

Program Output

© 2000 Prentice Hall, Inc.
All rights reserved.

```
#include<stdio.h>
struct student{
        char name[20];
        int rno;
        float gpa;
       };
      int main(){
        FILE *fp;
        int i,n;
        struct student s1;
        if(fp=fopen("students","w")==NULL)  {
         printf("\n Error in opening file");
         return;        }
        printf("Enter no of records");
        scanf("%d",&n);
        for(i=0;i<n;i++){
        printf("Enter name, rno and gpa");
        scanf("%s %d %f",s1.name,&s1.rno,&s1.gpa);
        fprintf(fp,"%s %d %f",s1.name,s1.rno,s1.gpa);
        }
        fflush(fp);
        printf("\n File written successfully");
        fclose(fp);
}
```

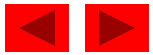
© 2000 Prentice Hall, Inc.
All rights reserved.

Three special streams

- Three special file streams are defined in the `<stdio.h>` header
  - stdin reads input from the keyboard
  - stdout send output to the screen
  - stderr prints errors to an error device (usually also the screen)

- What might this do?

```
fprintf (stdout,"Hello World!\n");
```



© 2000 Prentice Hall, Inc.
All rights reserved.

An example program

```
#include <stdio.h>
main()
{
 int i;

 fprintf(stdout,"Give value of i \n");
 fscanf(stdin,"%d",&i);
 fprintf(stdout,"Value of i=%d \n",i);
 fprintf(stderr,"No error: But an example to
 show error message.\n");
 }
```

```
Give value of i
15
Value of i=15
No error: But an example to show error message.
```

l, Inc.

Input File & Output File redirection

- One may redirect the standard input and standard output to other files (other than `stdin` and `stdout`).
- Usage: Suppose the executable file is `a.out`:

```
$ ./a.out <in.dat  >out.dat
```

`scanf()` will read data inputs from the file "`in.dat`", and `printf()` will output results on the file "`out.dat`".

```
$ ./a.out <in.dat  >>out.dat
```

`scanf()` will read data inputs from the file "`in.dat`", and `printf()` will append results at the end of the file "`out.dat`".

Iall, Inc.
All rights reserved.

Reading and Writing a character

- A character reading/writing is equivalent to reading/writing a byte.

```
int getchar( );
int putchar(int  c);
```

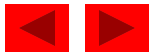
} stdin, stdout

```
int fgetc(FILE *fp);
int fputc(int c, FILE *fp);
```

} file

- Example:

```
char c;
c = getchar();
putchar(c);
```



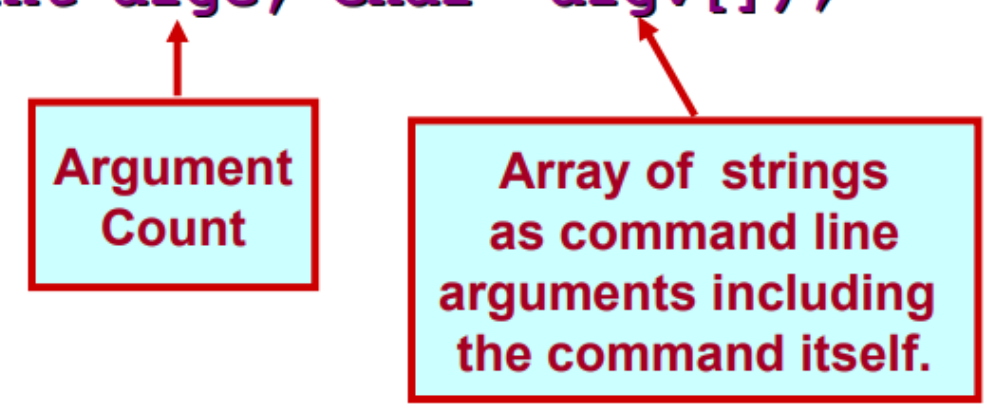
© 2000 Prentice Hall, Inc.
All rights reserved.

Command Line Arguments

How to access them?

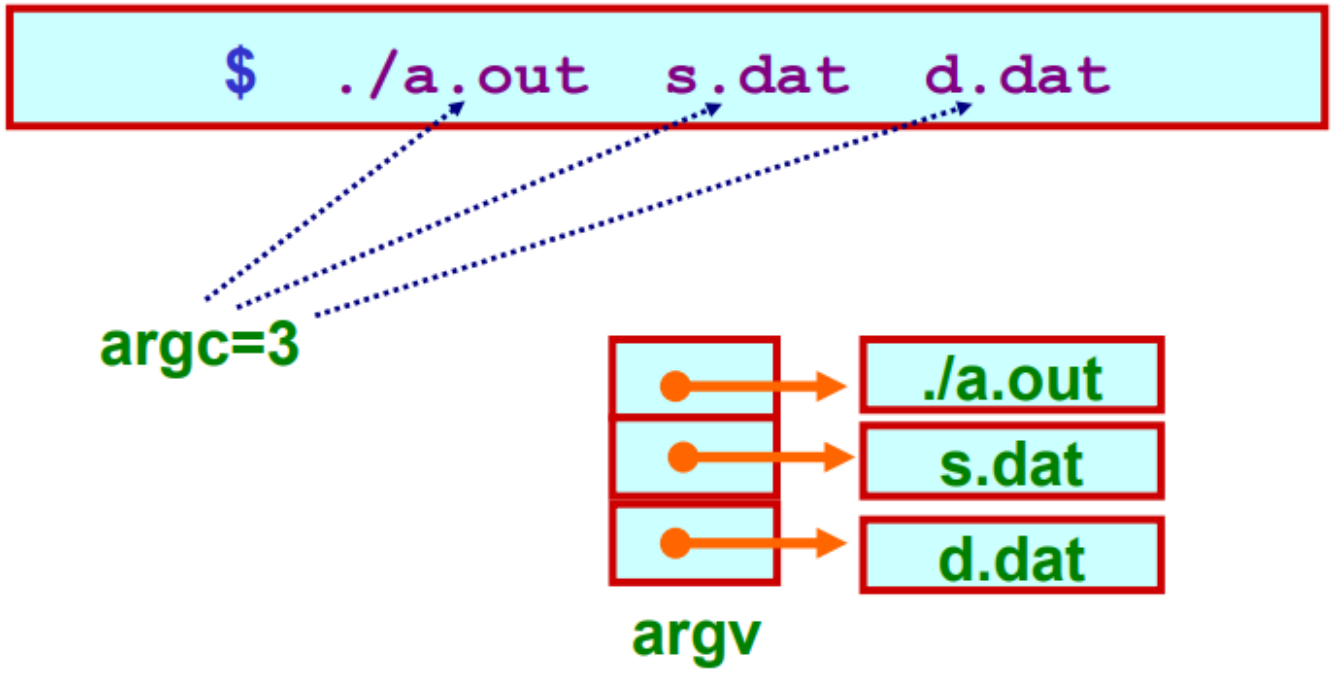
- Command line arguments may be passed by specifying them under `main()`.

```
int main (int argc, char *argv[]);
```



© 2000 Prentice Hall, Inc.
All rights reserved.

Example: Contd.

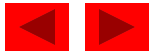


argv[0] = "./a.out" argv[1] = "s.dat" argv[2] = "d.dat"

© 2000 Prentice Hall, Inc.
All rights reserved.

Example: reading command line arguments

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

int main(int argc,char *argv[])
{
  FILE *ifp, *ofp;
  int i, c;
  char src_file[100],dst_file[100];

  if(argc!=3) {
    printf ("Usage: ./a.out <src_file> <dst_file> \n");
    exit(0);
  }
  else {
    strcpy (src_file, argv[1]);
    strcpy (dst_file, argv[2]);
  }
```



© 2000 Prentice Hall, Inc.
All rights reserved.

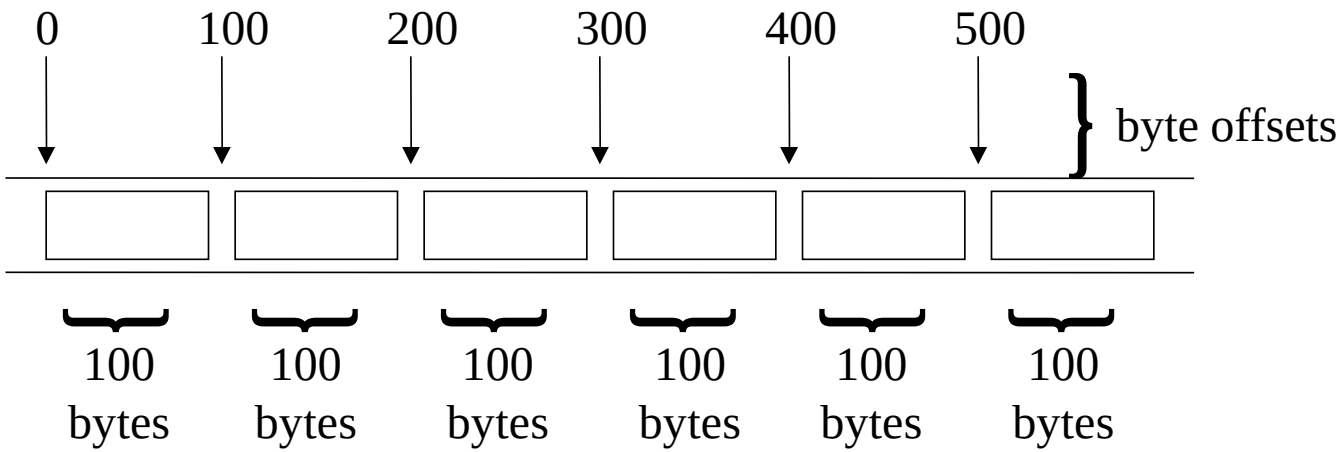
Example: contd.

```
   if ((ifp = fopen(src_file,"r")) == NULL)   {
      printf ("File does not exist.\n");
      exit(0);
   }

   if ((ofp = fopen(dst_file,"w")) == NULL)   {
      printf ("File not created.\n");
      exit(0);
   }

   while ((c = fgetc(ifp)) != EOF) {
      fputc (c,ofp);
   }

   fclose(ifp);
   fclose(ofp);
}
```

© 2000 Prentice Hall, Inc.
All rights reserved.

Random Access Files

- Random access files
  - Access individual records without searching through other records
  - Instant access to records in a file
  - Data can be inserted without destroying other data
  - Data previously stored can be updated or deleted without overwriting
- Implemented using fixed length records
  - Sequential files do not have fixed length records



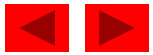
© 2000 Prentice Hall, Inc.
All rights reserved.

Creating a Random Access File

- Unformatted I/O functions
  - **`fwrite`**
    - Transfer bytes from a location in memory to a file
  - **`fread`**
    - Transfer bytes from a file to a location in memory
  - Example:

```
fwrite( &number, sizeof( int ), 1, myPtr );
```

    - **`&number`** – Location to transfer bytes from
    - **`sizeof( int )`** – Number of bytes to transfer
    - **`1`** – For arrays, number of elements to transfer
      - In this case, "one element" of an array is being transferred
    - **`myPtr`** – File to transfer to or from



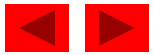
© 2000 Prentice Hall, Inc.
All rights reserved.

Creating a Random Access File

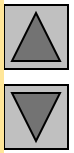
- Writing structs

```
fwrite( &myObject, sizeof (struct myStruct), 1, myPtr );
```

  - **`sizeof`** – returns size in bytes of object in parentheses
- To write several array elements
  - Pointer to array as first argument
  - Number of elements to write as third argument



© 2000 Prentice Hall, Inc.
All rights reserved.



Outline

1. Define struct

1.1 Initialize variable

1.2 Initialize struct

2. Open file

2.1 Write to file using unformatted output

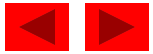
3. Close file

```
1  /* Fig. 11.11: fig11_11.c
2     Creating a randomly accessed file sequentially */
3  #include <stdio.h>
4
5  struct clientData {
6     int acctNum;
7     char lastName[ 15 ];
8     char firstName[ 10 ];
9     double balance;
10 };
11
12 int main()
13 {
14    int i;
15    struct clientData blankClient = { 0, "", "", 0.0 };
16    FILE *cfPtr;
17
18    if ( ( cfPtr = fopen( "credit.dat", "w" ) ) == NULL )
19       printf( "File could not be opened.\n" );
20    else {
21
22       for ( i = 1; i <= 100; i++ )
23          fwrite( &blankClient,
24                  sizeof( struct clientData ), 1, cfPtr );
25
26       fclose( cfPtr );
27    }
28
29    return 0;
30 }
```

© 2000 Prentice Hall, Inc.
All rights reserved.

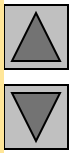
Writing Data Randomly to a Random Access File

- **`fseek`**
  - Sets file position pointer to a specific position
  - **`fseek(`** *pointer*, *offset*, *symbolic_constant* **`);`**
    - *pointer* – pointer to file
    - *offset* – file position pointer (0 is first location)
    - *symbolic_constant* – specifies where in file we are reading from
    - **`SEEK_SET`** – seek starts at beginning of file
    - **`SEEK_CUR`** – seek starts at current location in file
    - **`SEEK_END`** – seek starts at end of file



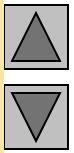
© 2000 Prentice Hall, Inc.
All rights reserved.

```c
/*
   Writing to a random access file */
#include <stdio.h>

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

int main()
{
   FILE *cfPtr;
   struct clientData client = { 0, "", "", 0.0 };

   if ( ( cfPtr = 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? " );
         fscanf( stdin, "%s%s%lf", client.lastName,
                 client.firstName, &client.balance );
         fseek( cfPtr, ( client.acctNum - 1 ) *
                 sizeof( struct clientData ), SEEK_SET );
         fwrite( &client, sizeof( struct clientData ), 1,
                  cfPtr );
         printf( "Enter account number\n? " );
```



Outline

© 2000 Prentice Hall, Inc.
All rights reserved.



Outline

```
33          scanf( "%d", &client.acctNum );
34       }
35
36       fclose( cfPtr );
37    }
38
39    return 0;
40 }
```

3. Close file

Program Output

```
Enter account number (1 to 100, 0 to end input)
? 37
Enter lastname, firstname, balance
? Barker Doug 0.00
Enter account number
? 29
Enter lastname, firstname, balance
? Brown Nancy -24.54
Enter account number
? 96
Enter lastname, firstname, balance
? Stone Sam 34.98
Enter account number
? 0
```

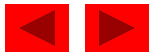
© 2000 Prentice Hall, Inc.
All rights reserved.

Reading Data Sequentially from a Random Access File

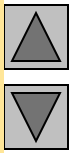
- **`fread`**
  - Reads a specified number of bytes from a file into memory

```
fread( &client, sizeof (struct clientData), 1, myPtr );
```

  - Can read several fixed-size array elements
    - Provide pointer to array
    - Indicate number of elements to read
  - To read multiple elements, specify in third argument



© 2000 Prentice Hall, Inc.
All rights reserved.



Outline

1. Define struct

1.1 Initialize variables

2. Read (`fread`)

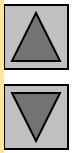
2.1 Print

```
1  /* Fig. 11.15: fig11_15.c
2     Reading a random access file sequentially */
3  #include <stdio.h>
4
5  struct clientData {
6     int acctNum;
7     char lastName[ 15 ];
8     char firstName[ 10 ];
9     double balance;
10 };
11
12 int main()
13 {
14    FILE *cfPtr;
15    struct clientData client = { 0, "", "", 0.0 };
16
17    if ( ( cfPtr = fopen( "credit.dat", "r" ) ) == NULL )
18       printf( "File could not be opened.\n" );
19    else {
20       printf( "%-6s%-16s%-11s%10s\n", "Acct", "Last Name",
21               "First Name", "Balance" );
22
23       while ( !feof( cfPtr ) ) {
24          fread( &client, sizeof( struct clientData ), 1,
25                 cfPtr );
26
27          if ( client.acctNum != 0 )
28             printf( "%-6d%-16s%-11s%10.2f\n",
29                     client.acctNum, client.lastName,
30                     client.firstName, client.balance );
31       }
32
```

© 2000 Prentice Hall, Inc.
All rights reserved.

```
33       fclose( cfPtr );
34    }
35
36    return 0;
37 }
```

```
Acct  Last Name       First Name    Balance
29    Brown           Nancy          -24.54
33    Dunn            Stacey         314.33
37    Barker          Doug             0.00
88    Smith           Dave           258.34
96    Stone           Sam             34.98
```



Outline

3. Close file

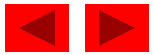
Program Output

Thank You!

© 2000 Prentice Hall, Inc.
All rights reserved.

What is Bitwise Structure?

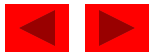
- The smallest type is of 8 bits (char).
- Sometimes we need only a single bit.
- For instance, storing the status of the lights in 8 rooms:
  - We need to define an array of at least 8 chars.
    If the light of room 3 is turned on the value of
    the third char is 1, otherwise 0.
  - Total array of 64 bits.



© 2000 Prentice Hall, Inc.
All rights reserved.

What is Bitwise Structure?

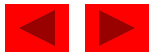
- It is better to define only 8 bits since a bit can also store the values 0 or 1.
- But the problem is that there is no C type which is 1 bit long (char is the longer with 1 byte).
- Solution: define a char (8 bits) but refer to each bit separately.
- `Bitwise` operators, introduced by the C language, provide one of its more powerful tools for using and manipulating memory. They give the language the real power of a “low-level language”.



© 2000 Prentice Hall, Inc.
All rights reserved.

What is Bitwise Structure?

- Accessing bits directly is fast and efficient, especially if you are writing a real-time application.
- A single bit cannot be accessed directly, since it has no address of its own.
- The language introduces the **`bitwise`** operators, which help in manipulating a single bit of a byte.
- **`bitwise`** operators may be used on integral types only (unsigned types are preferable).



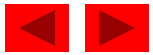
© 2000 Prentice Hall, Inc.
All rights reserved.

Bitwise Operators

&	bitwise AND
|	bitwise OR
^	bitwise XOR
~	1's compliment
<<	Shift left
>>	Shift right

All these operators can be suffixed with = (assignment)

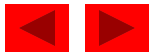
For instance a &= b; is the same as a = a & b;



© 2000 Prentice Hall, Inc.
All rights reserved.

Bitwise Operators – truth table

a	b	a&b	a|b	a^b	~a
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0



© 2000 Prentice Hall, Inc.
All rights reserved.

Bitwise Operators - Examples

```
11010011
&
10001100
------------
10000000
```

```
11010011
|
10001100
------------
11011111
```

```
11010011
^
10001100
------------
01011111
```

```
~11010011
------------
 00101100
```

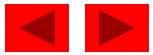
```
11010011>>3
------------
 00011010
```

```
11010011<<3
------------
 10011000
```


© 2000 Prentice Hall, Inc.
All rights reserved.

Setting Bits

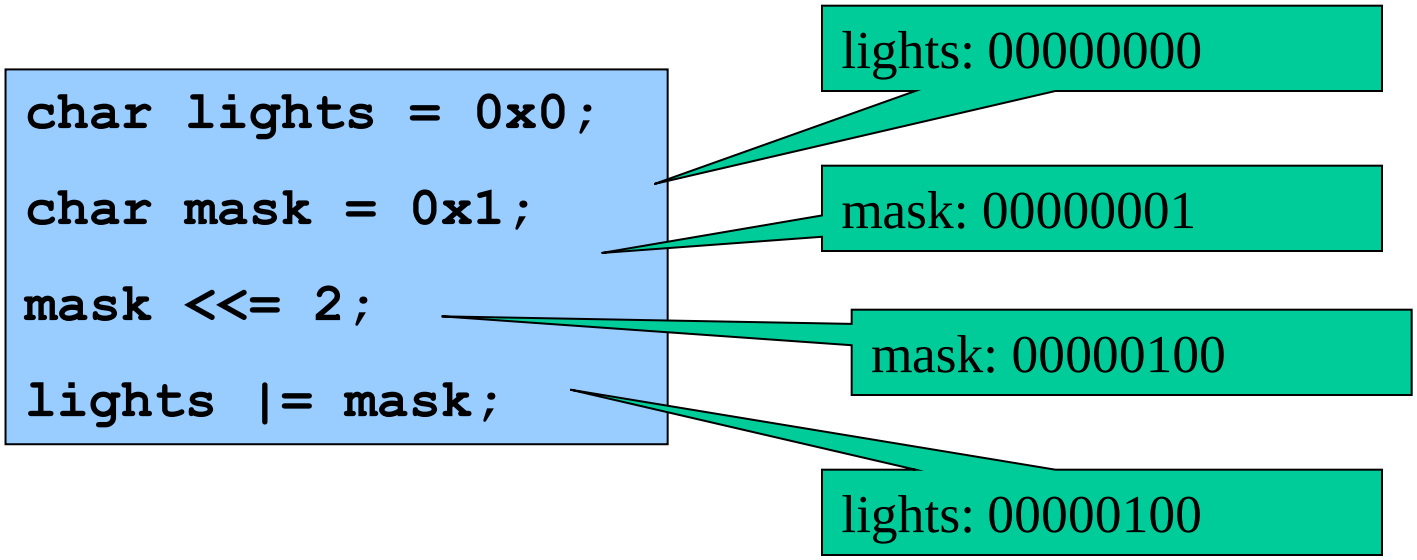
- How can we set a bit on or off?
- Manipulations on bits are enabled by mask and bitwise operators.
- Bitwise OR of anything with 1 results in 1.
- Bitwise AND of anything with 0 results in 0.



© 2000 Prentice Hall, Inc.
All rights reserved.

Setting Bits

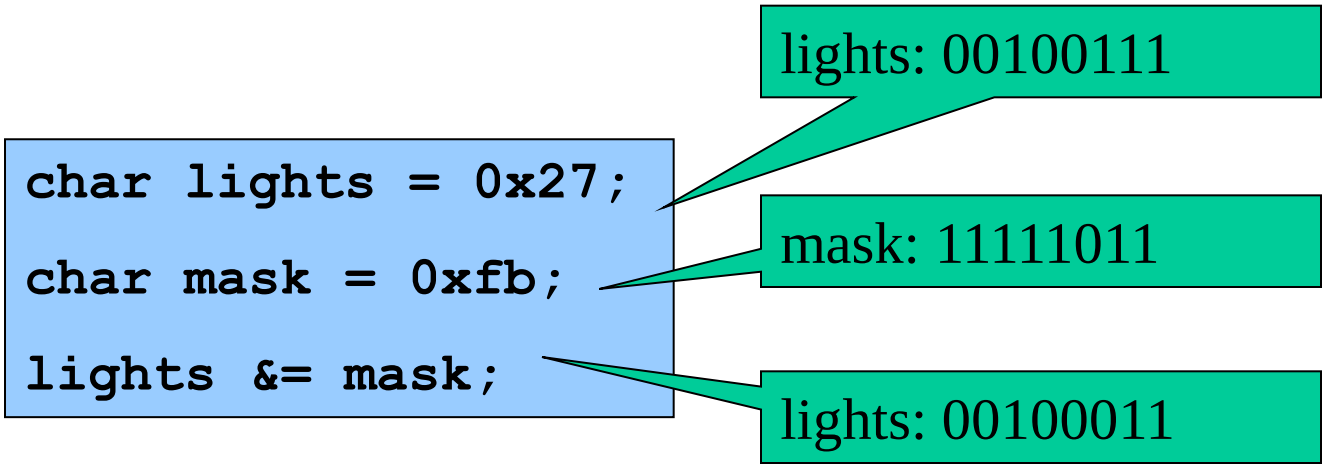
- For instance, how can we turn on the light in room #3?



© 2000 Prentice Hall, Inc.
All rights reserved.

Setting Bits

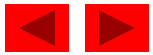
- For instance, how can we turn off the light in room #3?



© 2000 Prentice Hall, Inc.
All rights reserved.

Getting Bits (Testing bits)

- How can we know if a bit is on or off?
- Manipulations on bits are enabled by mask and bitwise operators.
- Bitwise AND of anything with 1 results in the same value.



© 2000 Prentice Hall, Inc.
All rights reserved.

Getting Bits

- For instance, how can we check if the light in room #3 is turned on or off?

```
char lights = 0x27;
char mask = 0x1;
mask <<= 2;
if(lights & mask)
  puts("turned on");
else
  puts("turned off");
```

lights: 00100111

mask: 00000001

mask: 00000100

lights & mask: 00000100

© 2000 Prentice Hall, Inc.
All rights reserved.

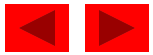
Inverting & Clearing Bits

- Inverting bits: XOR (^)
- XOR with mask bit 1 → the bit is inverted
- XOR with mask bit 0 → the bit is unchanged

- Clearing bits → &~ with mask bit 1

Hence:

- Set – OR
- Test - &



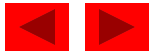
© 2000 Prentice Hall, Inc.
All rights reserved.

The Preprocessor

© 2000 Prentice Hall, Inc.
All rights reserved.

Introduction

- Preprocessing
  - Occurs before program compiled
    - Inclusion of external files
    - Definition of symbolic constants
    - Macros
    - Conditional compilation
  - All directives begin with **#**
    - Can only have whitespace before directives



© 2000 Prentice Hall, Inc.
All rights reserved.

Preprocessor

- Preprocessor processes source program before it is passed to compiler.

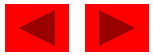
- Produce a source code file with the preprocessing commands properly sorted out.

© 2000 Prentice Hall, Inc.
All rights reserved.

Preprocessor Directives

- Preprocessor commands are known as ***directives.***
- Preprocessor provides certain ***features***.
- These features are also known as ***preprocessor directives***.
- Preprocessor directives start with # sign.

```
#include <stdio.h>
```



© 2000 Prentice Hall, Inc.
All rights reserved.

Preprocessor Directives...

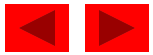
- Preprocessor directives can be placed any where in the source program.
- **Note: Place it at start of the program.**
- Each preprocessor directive must be on it's own line.

```
#include <stdio.h>
#include <conio.h>
```

© 2000 Prentice Hall, Inc.
All rights reserved.

Preprocessor Directives

- Macro Expansion
- File Inclusion
- Conditional compilation
- Miscellaneous directives



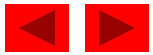
© 2000 Prentice Hall, Inc.
All rights reserved.

Macro Expansion

- ***#define*** directive is known as macro expansion.
- Definition:

```
#define PI 3.1415
```

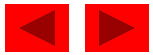
- ***General Form:***
  - **#define macro_template macro_expansion**
  - **#define macro_name char_sequence**



© 2000 Prentice Hall, Inc.
All rights reserved.

Macro Expansion...

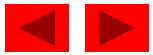
- Preprocessor search for macro definition.
- After finding ***#define directive*** it search entire program for ***macro_template***.
- Replace each ***macro_template*** with ***macro_expansion***.
- **Best Practice:** Use capital letters for macro template.
- Do not use semicolon '***;***'



© 2000 Prentice Hall, Inc.
All rights reserved.

The #define Preprocessor Directive: Macros

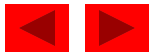
- Macro
  - Operation specified in **`#define`**
  - Intended for legacy C programs
  - Macro without arguments
    - Treated like a symbolic constant
  - Macro with arguments
    - Arguments substituted for replacement text
    - Macro expanded
  - Performs a text substitution
    - No data type checking



© 2000 Prentice Hall, Inc.
All rights reserved.

Macros, Why?

- To write efficient programs.
- To increase readiabiality of programs.
- Variable vs macro_template
  - Compiler can generate faster and compact code for constant than it can for variables.
  - When you are dealing with a constant, why use variable.
  - A variable may change in the program.



© 2000 Prentice Hall, Inc.
All rights reserved.

Macros, Where?

- Replace operator:

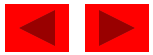
```
#define AND &&
#define OR ||
```

- Replace condition:

```
#define EXCELLENT (a>=75)
```

- Replace statement:

```
#define ALERT printf("Security Alert");
```



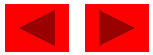
© 2000 Prentice Hall, Inc.
All rights reserved.

Macros...

- Defined macro name can be used as a part of definition of other macro name.

```
#define MIN 1
#define MAX 9
#define MIDDLE (MAX-MIN)/2
```

- No text substitution occur if the identifier is within a quoted string.



© 2000 Prentice Hall, Inc.
All rights reserved.

Macros with Arguments

- Macros can have arguments, same as functions

```
#define ISEXCELLENT(x) (x >= 75)
#define ISLOWER(x) (x>=97 && x<=122)
```

- **Note: Space between Macro and it's arguments.**

ISLOWER(x) **ISLC R (x)**

© 2000 Prentice Hall, Inc.
All rights reserved.

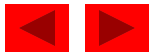
Macros with Arguments...

- Macros expansions should be enclosed within paranthesis.

```
#define ISLOWER(x) (x>=97 && x<=122)
if(!ISLOWER('a'));
```

- Use '\' to split macro in multiple line.

```
#define HLINE for(i=0; i < 40; i++)\
                printf("_");
```



© 2000 Prentice Hall, Inc.
All rights reserved.

The #define Preprocessor Directive: Macros

- Example

```
#define CIRCLE_AREA( x ) ( PI * ( x ) * ( x ) )
area = CIRCLE_AREA( 4 );
```

becomes

```
area = ( 3.14159 * ( 4 ) * ( 4 ) );
```

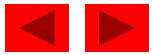
- Use parentheses
  - Without them,

```
#define CIRCLE_AREA( x )  PI * x * x
area = CIRCLE_AREA( c + 2 );
```

becomes

```
area = 3.14159 * c + 2 * c + 2;
```

which evaluates incorrectly



© 2000 Prentice Hall, Inc.
All rights reserved.

The #define Preprocessor Directive: Macros

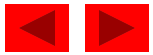
- Multiple arguments

```
#define RECTANGLE_AREA( x, y )  ( ( x ) * ( y ) )
rectArea = RECTANGLE_AREA( a + 4, b + 7 );
```

  becomes

```
rectArea = ( ( a + 4 ) * ( b + 7 ) );
```

- **#undef**
  - Undefines symbolic constant or macro
  - Can later be redefined

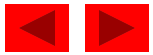


© 2000 Prentice Hall, Inc.
All rights reserved.

Macros vs Functions

MACROS	FUNCTIONS
Just the replacement of the code.	Passing arguments, doing calculation, returning results. (More serious work).
Macros make the program run faster.	Function calls and return make the program slow.
Increase the program size	Make program smaller and compact.

….more



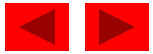
© 2000 Prentice Hall, Inc.
All rights reserved.

File Inclusion

- causes **one file to be included in another**.

```
#include <filename> //OR
#include “filename”
```

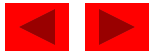
- ***<filename> :*** search the directory on current directory only.
- ***“filename” :*** search the directory on current directory and specified directories as specified in the include search path.



© 2000 Prentice Hall, Inc.
All rights reserved.

Why File Inclusion?

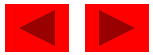
- Divide a program in multiple files.
  - Each file contains related functions.
- Some functions or macros are required in each program
  - Put them in a file (Library).
  - Include them in program that need them.
- **Nested includes:** Included file may have more included files in it.



© 2000 Prentice Hall, Inc.
All rights reserved.

Conditional Compilation

- Write single program to run on different environments.
  - **#ifdef** – if defined
  - **#endif** – end if
  - **#else** – else
  - **#ifndef** – in not defined
  - **#if** – if
  - **#elif** – else if



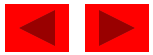
© 2000 Prentice Hall, Inc.
All rights reserved.

Conditional Compilation

- Control preprocessor directives and compilation
- Structure similar to **`if`**

```
#if !defined( NULL )
   #define NULL 0
#endif
```

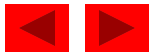
  - Determines if symbolic constant **`NULL`** defined
  - If **`NULL`** defined,
    - **`defined( NULL )`** evaluates to **`1`**
    - **`#define`** statement skipped
  - Otherwise
    - **`#define`** statement used
  - Every **`#if`** ends with **`#endif`**



© 2000 Prentice Hall, Inc.
All rights reserved.

Conditional Compilation

- Can use else
  - **`#else`**
  - **`#elif`** is "else if"
- Abbreviations
  - **`#ifdef`** short for
    - **`#if defined(name)`**
  - **`#ifndef`** short for
    - **`#if !defined(name)`**



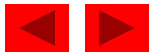
© 2000 Prentice Hall, Inc.
All rights reserved.

#ifdef & #endif

- ***General form:***

```
#ifdef macroname
        statement sequence
#endif
```

- if macroname has been defined using **#define** the code between **#ifdef** & **#endif** will execute.

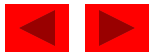


© 2000 Prentice Hall, Inc.
All rights reserved.

#else

- Use #else with #ifdef same as else with if.
- ***General-form:***

```
#ifdef macroname
        statement sequence
#else
        statement sequence
#endif
```



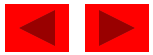
© 2000 Prentice Hall, Inc.
All rights reserved.

#ifndef

- #ifndef is just opposite to #ifdef.

```
#ifndef __file_h
        #define __file_h
```

- #if directive test whether an expression evaluates to nonzero value or not.
- #elif used same as else if.



© 2000 Prentice Hall, Inc.
All rights reserved.

The # and ## Operators

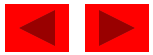
- **#** operator (stringizing)
  - Replacement text token converted to string with quotes

```
#define HELLO( x ) cout << "Hello, " #x << endl;
```

  - **HELLO(JOHN)** becomes
    - `cout << "Hello, " "John" << endl;`
    - Same as `cout << "Hello, John" << endl;`
- **##** operator
  - Concatenates two tokens

```
#define TOKENCONCAT( x, y )  x ## y
```

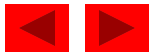
  - **TOKENCONCAT(O, K)** becomes
    - **OK**



© 2000 Prentice Hall, Inc.
All rights reserved.

Line Numbers

- **`#line`**
  - Renumbers subsequent code lines, starting with integer
    - **`#line 100`**
  - File name can be included
  - **`#line 100 "file1.cpp"`**
    - Next source code line is numbered **`100`**
    - For error purposes, file name is **`"file1.cpp"`**
    - Can make syntax errors more meaningful
    - Line numbers do not appear in source file

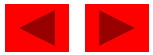


© 2000 Prentice Hall, Inc.
All rights reserved.

Predefined Symbolic Constants

- Five predefined symbolic constants
  - Cannot be used in **`#define`** or **`#undef`**

Symbolic constant	Description
`__LINE__`	The line number of the current source code line (an integer constant).
`__FILE__`	The presumed name of the source file (a string).
`__DATE__`	The date the source file is compiled (a string of the form **`"Mmm dd yyyy"`** such as **`"Jan 19 2001"`**).
`__TIME__`	The time the source file is compiled (a string literal of the form **`"hh:mm:ss"`**).

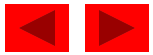


© 2000 Prentice Hall, Inc.
All rights reserved.

Where conditional compilation?

- Run the same code on different environment.

```
#ifdef WINDOWS
        statement sequence
#else
        statemet sequence
#endif
```



© 2000 Prentice Hall, Inc.
All rights reserved.

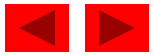
Where conditional compilation?

- To avoide multiple declaration error.

© 2000 Prentice Hall, Inc.
All rights reserved.

The #error and #pragma Preprocessor Directives

- **`#error` *`tokens`***
  - Prints implementation-dependent message
  - Tokens are groups of characters separated by spaces
    - **`#error 1 - Out of range error`** has 6 tokens
  - Compilation may stop (depends on compiler)
- **`#pragma` *`tokens`***
  - Actions depend on compiler
  - May use compiler-specific options
  - Unrecognized **`#pragma`**s are ignored



© 2000 Prentice Hall, Inc.
All rights reserved.