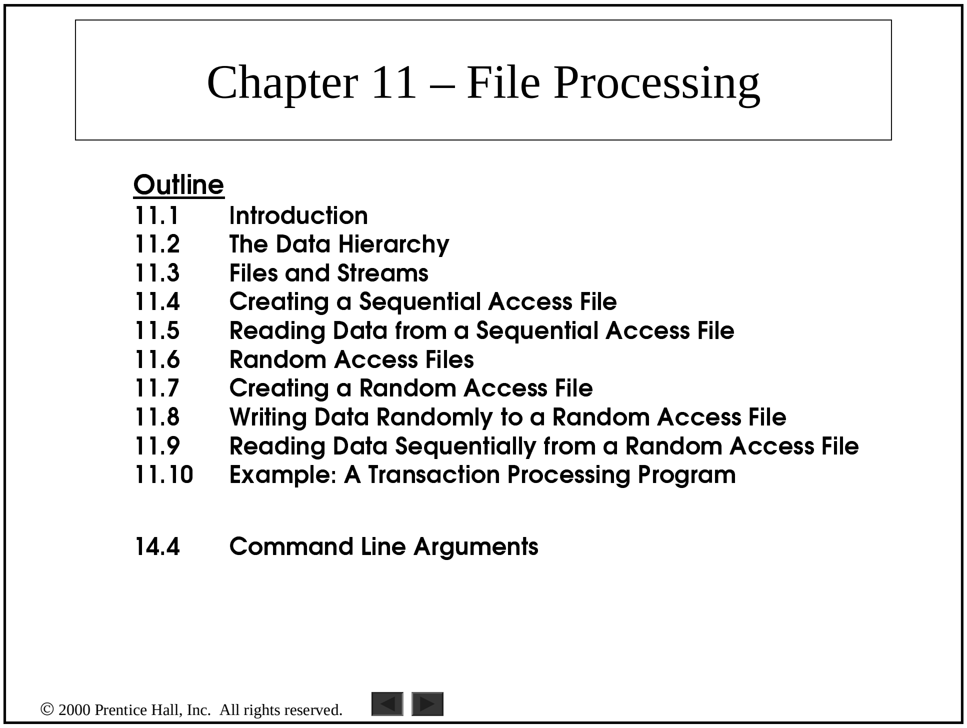


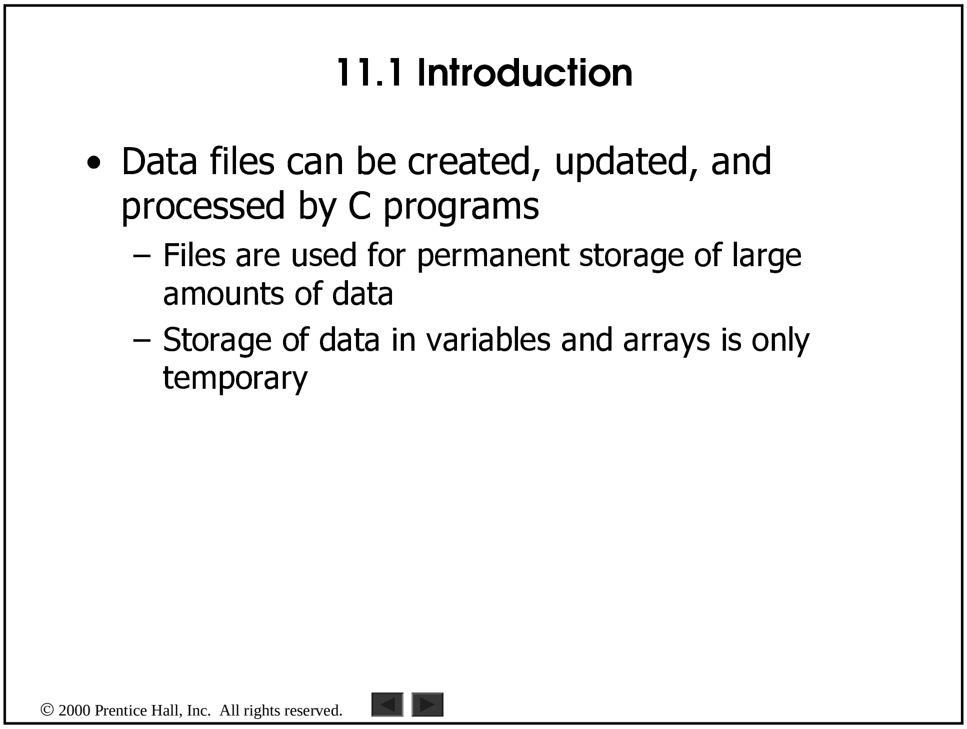
Chapter 11 – File Processing

Outline

11.1 Introduction
11.2 The Data Hierarchy
11.3 Files and Streams
11.4 Creating a Sequential Access File
11.5 Reading Data from a Sequential Access File
11.6 Random Access Files
11.7 Creating a Random Access File
11.8 Writing Data Randomly to a Random Access File
11.9 Reading Data Sequentially from a Random Access File
11.10 Example: A Transaction Processing Program

14.4 Command Line Arguments

◀ ▶



11.1 Introduction

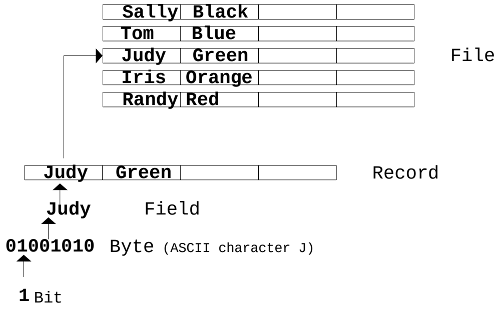
- Data files can be created, updated, and processed by C programs
  - Files are used for permanent storage of large amounts of data
  - Storage of data in variables and arrays is only temporary

◀ ▶

11.2 The Data Hierarchy

- Bit - smallest data item
  - Value of **`0`** or **`1`**
- Byte – 8 bits
  - Used to store a character
    - Decimal digits, letters, and special symbols
- Field - group of characters conveying meaning
  - Example: your name
- Record – group of related fields
  - Represented by a **`struct`** or a **`class`**
  - Example: In a payroll system, a record for a particular employee that contained his/her identification number, name, address, etc.
- File – group of related records
  - Example: payroll file
- Database – group of related files

11.2 The Data Hierarchy (II)



Sally **Black**
Tom **Blue**
Judy **Green**
Iris **Orange**
Randy **Red**

File

Judy **Green**

Record

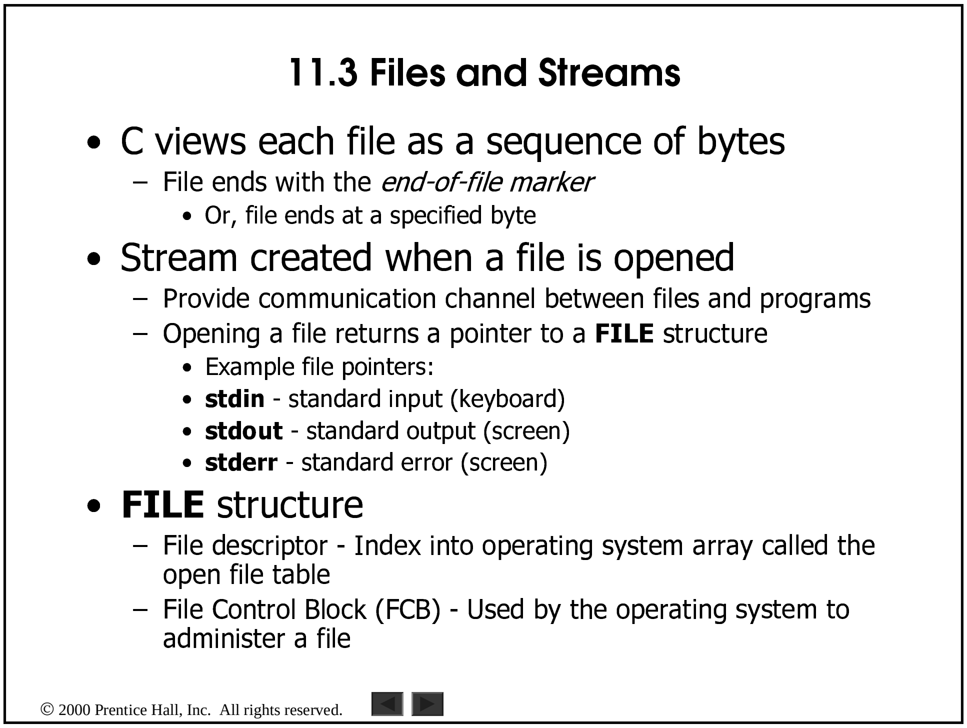
Judy

Field

01001010 Byte (ASCII character J)

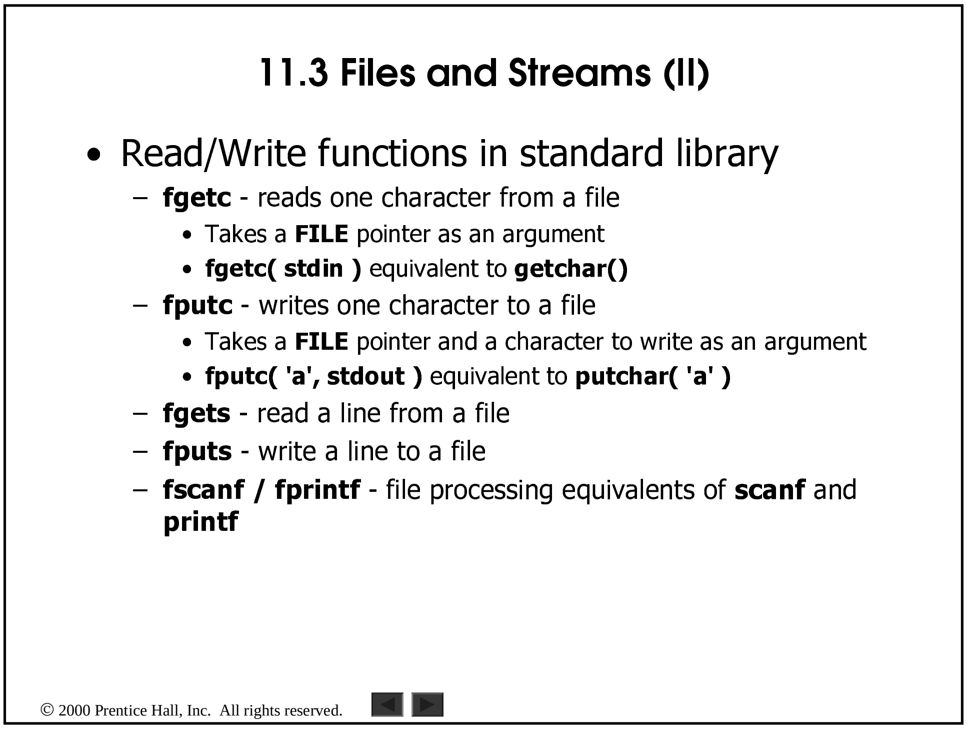
1 Bit

- Record key
  - Identifies a record to facilitate the retrieval of specific records from a file
- Sequential file
  - Records typically sorted by key



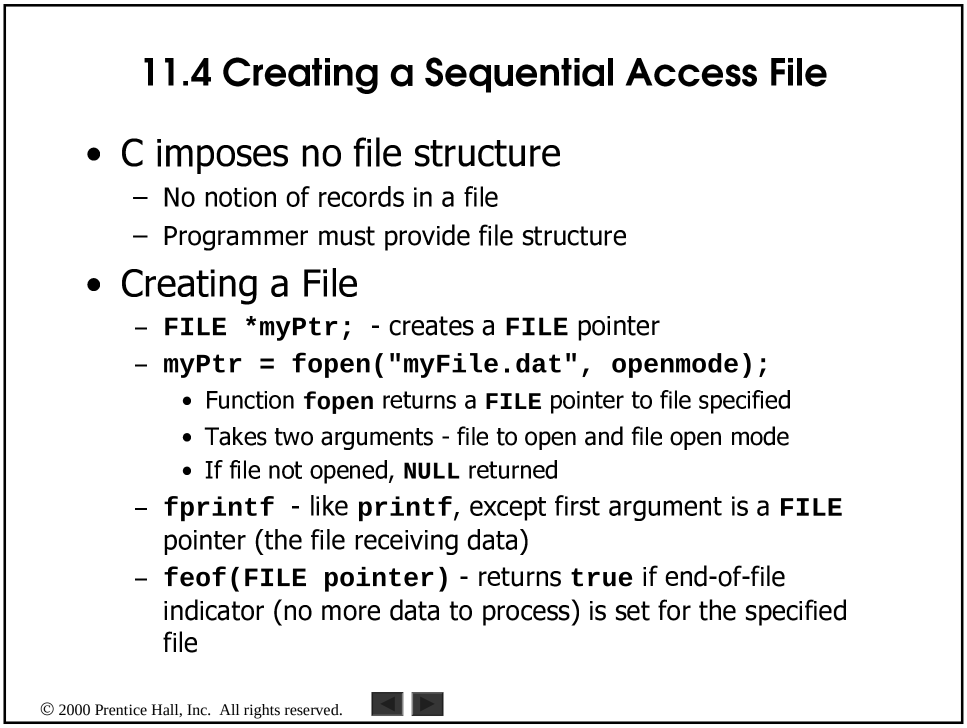
11.3 Files and Streams

- C views each file as a sequence of bytes
  - File ends with the *end-of-file marker*
    - Or, file ends at a specified byte
- Stream created when a file is opened
  - Provide communication channel between files and programs
  - Opening a file returns a pointer to a **FILE** structure
    - Example file pointers:
    - **stdin** - standard input (keyboard)
    - **stdout** - standard output (screen)
    - **stderr** - standard error (screen)
- **FILE** structure
  - File descriptor - Index into operating system array called the open file table
  - File Control Block (FCB) - Used by the operating system to administer a file



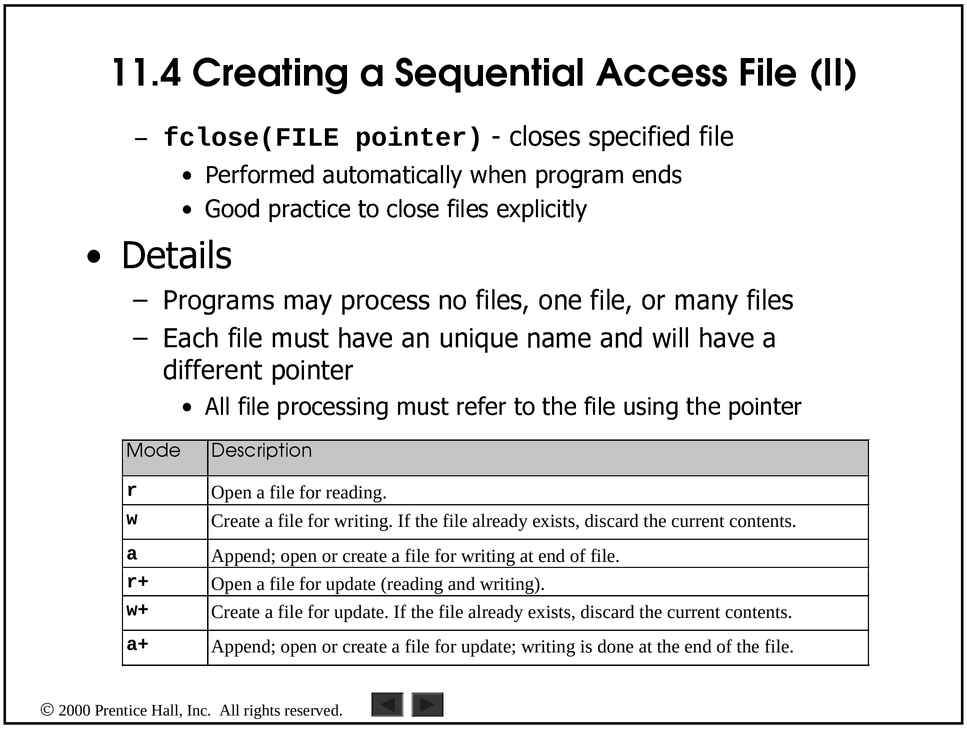
11.3 Files and Streams (II)

- Read/Write functions in standard library
  - **fgetc** - reads one character from a file
    - Takes a **FILE** pointer as an argument
    - **fgetc(stdin)** equivalent to **getchar()**
  - **fputc** - writes one character to a file
    - Takes a **FILE** pointer and a character to write as an argument
    - **fputc('a', stdout)** equivalent to **putchar('a')**
  - **fgets** - read a line from a file
  - **fputs** - write a line to a file
  - **fscanf / fprintf** - file processing equivalents of **scanf** and **printf**



11.4 Creating a Sequential Access File

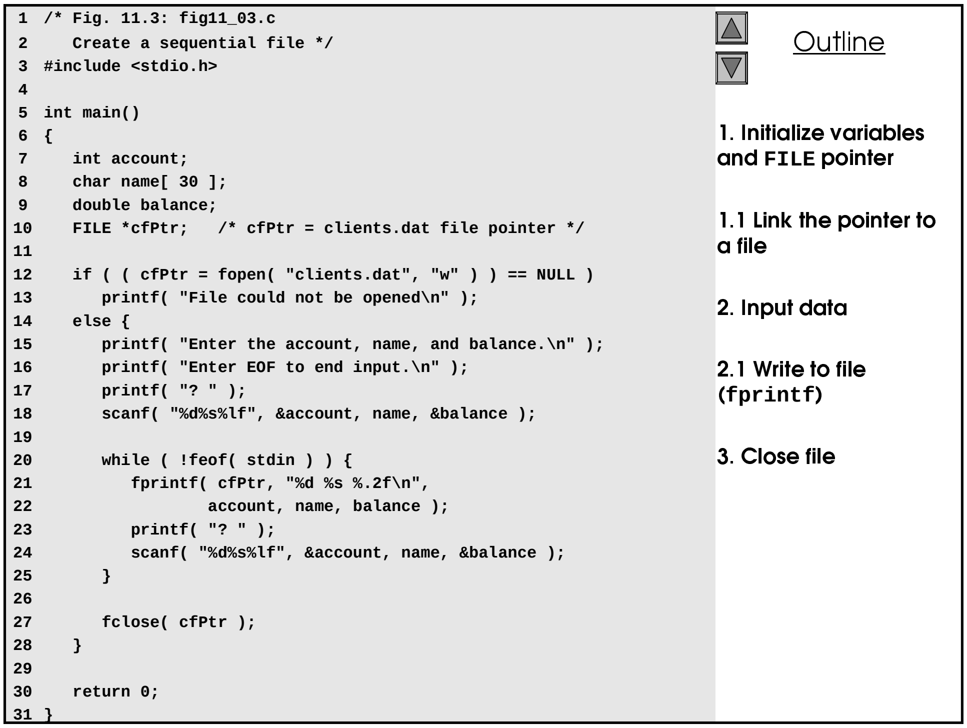
- C imposes no file structure
  - No notion of records in a file
  - Programmer must provide file structure
- Creating a File
  - **`FILE *myPtr;`** - creates a **`FILE`** pointer
  - **`myPtr = fopen("myFile.dat", openmode);`**
    - Function **`fopen`** returns a **`FILE`** pointer to file specified
    - Takes two arguments - file to open and file open mode
    - If file not opened, **`NULL`** returned
  - **`fprintf`** - like **`printf`**, except first argument is a **`FILE`** pointer (the file receiving data)
  - **`feof(FILE pointer)`** - returns **`true`** if end-of-file indicator (no more data to process) is set for the specified file



11.4 Creating a Sequential Access File (II)

- **`fclose(FILE pointer)`** - closes specified file
  - Performed automatically when program ends
  - Good practice to close files explicitly
- Details
  - Programs may process no files, one file, or many files
  - Each file must have an unique name and will have a different pointer
    - All file processing must refer to the file using the pointer

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.



```
/* Fig. 11.3: fig11_03.c
   Create a sequential file */
#include <stdio.h>

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

   if ( ( cfPtr = fopen( "clients.dat", "w" ) ) == NULL )
      printf( "File could not be opened\n" );
   else {
      printf( "Enter the account, name, and balance.\n" );
      printf( "Enter EOF to end input.\n" );
      printf( "? " );
      scanf( "%d%s%lf", &account, name, &balance );

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

      fclose( cfPtr );
   }

   return 0;
}
```

Outline

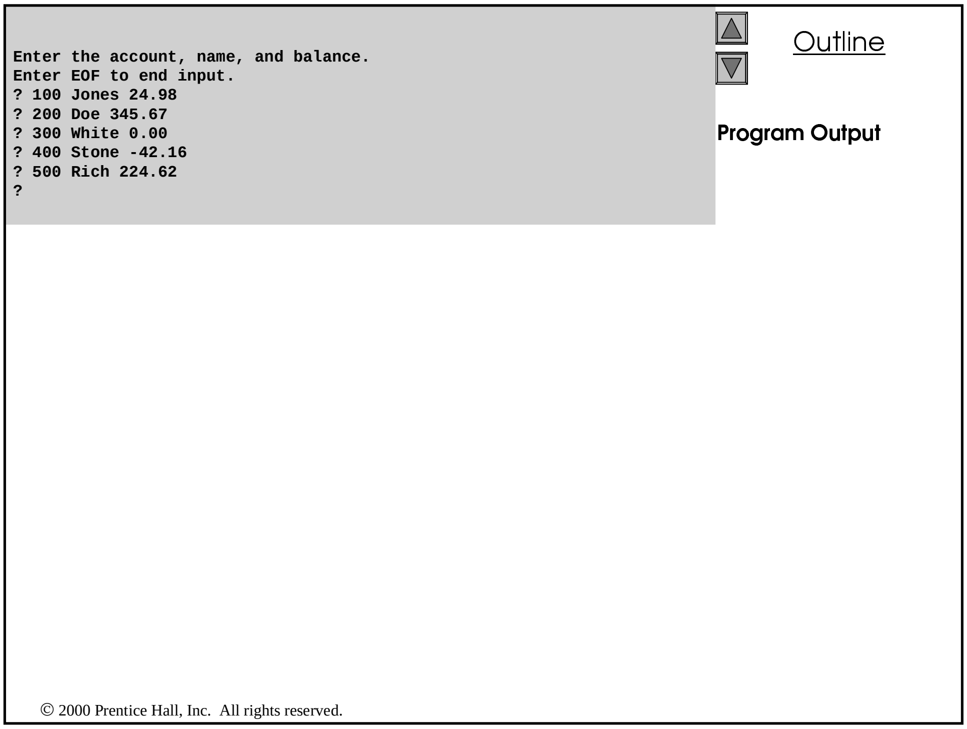
1. Initialize variables and `FILE` pointer

1.1 Link the pointer to a file

2. Input data

2.1 Write to file (`fprintf`)

3. Close file



```
Enter the account, name, and balance.
Enter EOF to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
?
```

Outline

Program Output

14.4 Using Command-Line Arguments

- Pass arguments to **main** in DOS and UNIX

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

  int argc - number of arguments passed

  char *argv[] - array of strings, has names of arguments in order (**argv[0]** is first argument)

  Example: **$ copy input output**

  argc: 3

  argv[0]: "copy"

  argv[1]: "input"

  argv[2]: "output"

```
/* Fig. 14.3: fig14_03.c
   Using command-line arguments */
#include <stdio.h>

int main( int argc, char *argv[] )
{
   FILE *inFilePtr, *outFilePtr;
   int c;

   if ( argc != 3 )
      printf( "Usage: copy infile outfile\n" );
   else
      if ( ( inFilePtr = fopen( argv[ 1 ], "r" ) )

         if ( ( outFilePtr = fopen( argv[ 2 ], "w" ) ) != NULL )

            while ( ( c = fgetc( inFilePtr ) ) != EOF )
               fputc( c, outFilePtr );

         else
            printf( "File \"%s\" could no

      else
         printf( "File \"%s\" could not be opened\n", argv[ 1 ] );

   return 0;
}
```

Notice **argc** and **argv[]** in **main**

argv[1] is the second argument, and is being read.

argv[2] is the third argument, and is being written to.

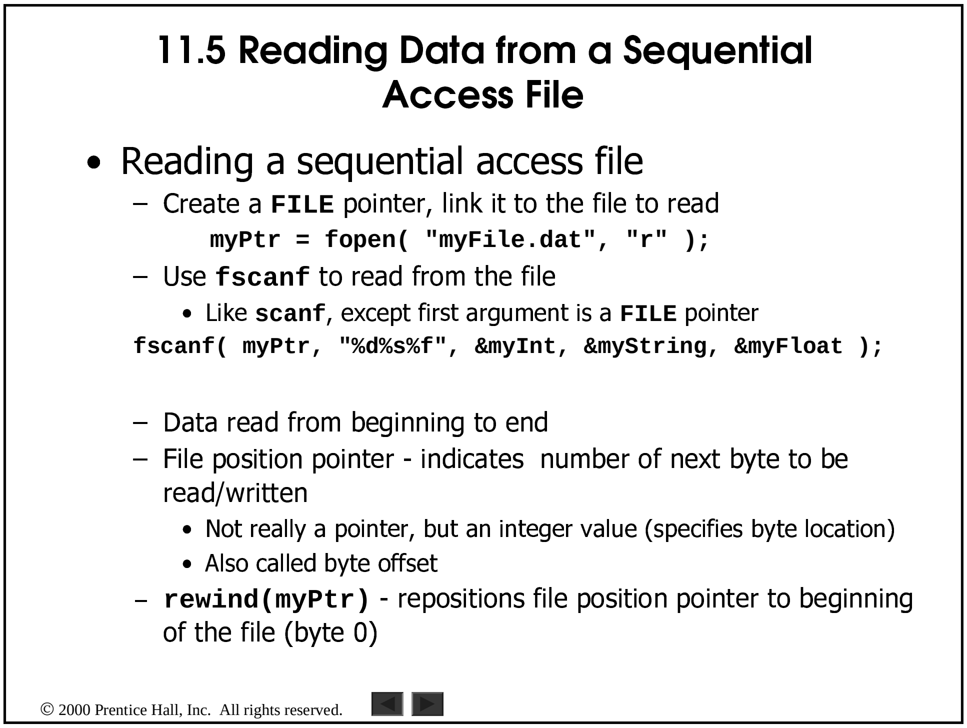
Loop until **End Of File**. **fgetc** a character from **inFilePtr** and **fputc** it into **outFilePtr**.

Outline

1. Initialize variables

y open type
rite)

3. Copy file

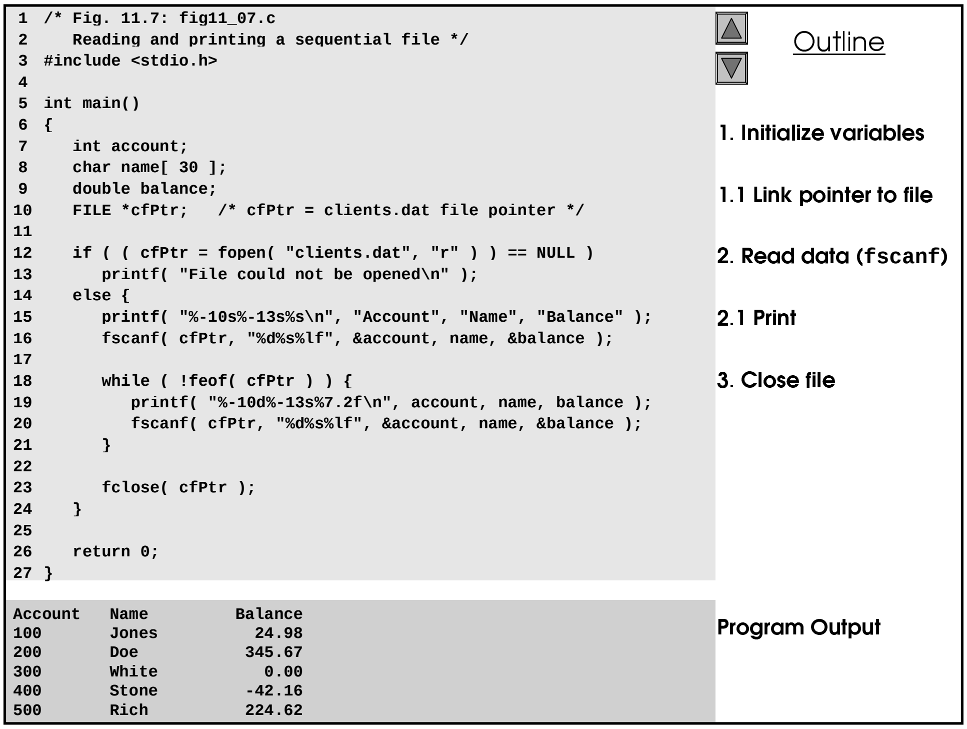


11.5 Reading Data from a Sequential Access File

- Reading a sequential access file
  - Create a **FILE** pointer, link it to the file to read

    myPtr = fopen("myFile.dat", "r");
  - Use **fscanf** to read from the file
    - Like **scanf**, except first argument is a **FILE** pointer

    fscanf(myPtr, "%d%s%f", &myInt, &myString, &myFloat);
  - Data read from beginning to end
  - File position pointer - indicates number of next byte to be read/written
    - Not really a pointer, but an integer value (specifies byte location)
    - Also called byte offset
  - **rewind(myPtr)** - repositions file position pointer to beginning of the file (byte 0)



```
/* Fig. 11.7: fig11_07.c
   Reading and printing a sequential file */
#include <stdio.h>

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

   if ( ( cfPtr = fopen( "clients.dat", "r" ) ) == NULL )
      printf( "File could not be opened\n" );
   else {
      printf( "%-10s%-13s%s\n", "Account", "Name", "Balance" );
      fscanf( cfPtr, "%d%s%lf", &account, name, &balance );

      while ( !feof( cfPtr ) ) {
         printf( "%-10d%-13s%7.2f\n", account, name, balance );
         fscanf( cfPtr, "%d%s%lf", &account, name, &balance );
      }

      fclose( cfPtr );
   }

   return 0;
}
```

Outline

1. Initialize variables

1.1 Link pointer to file

2. Read data (fscanf)

2.1 Print

3. Close file

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

Program Output

```
/* Fig. 11.8: fig11_08.c
   Credit inquiry program */
#include <stdio.h>

int main()
{
   int request, account;
   double balance;
   char name[ 30 ];
   FILE *cfPtr;

   if ( ( cfPtr = fopen( "clients.dat", "r" ) ) == NULL )
      printf( "File could not be opened\n" );
   else {
      printf( "Enter request\n"
             " 1 - List accounts with zero balances\n"
             " 2 - List accounts with credit balances\n"
             " 3 - List accounts with debit balances\n"
             " 4 - End of run\n? " );
      scanf( "%d", &request );

      while ( request != 4 ) {
         fscanf( cfPtr, "%d%s%lf", &account, name,
                 &balance );

         switch ( request ) {
            case 1:
               printf( "\nAccounts with zero "
                       "balances:\n" );

               while ( !feof( cfPtr ) ) {

```

Outline

1. Initialize variables

2. Open file

2.1 Input choice

2.2 Scan files

3. Print

```
                  if ( balance == 0 )
                     printf( "%-10d%-13s%7.2f\n",
                            account, name, balance );

                  fscanf( cfPtr, "%d%s%lf",
                         &account, name, &balance );
               }

               break;
            case 2:
               printf( "\nAccounts with credit "
                       "balances:\n" );

               while ( !feof( cfPtr ) ) {

                  if ( balance < 0 )
                     printf( "%-10d%-13s%7.2f\n",
                            account, name, balance );

                  fscanf( cfPtr, "%d%s%lf",
                         &account, name, &balance );
               }

               break;
            case 3:
               printf( "\nAccounts with debit "
                       "balances:\n" );

               while ( !feof( cfPtr ) ) {

                  if ( balance > 0 )
                     printf( "%-10d%-13s%7.2f\n",
```

Outline

2.2 Scan files

3. Print

```
                            account, name, balance );

                  fscanf( cfPtr, "%d%s%lf",
                         &account, name, &balance );
               }

               break;
         }

         rewind( cfPtr );
         printf( "\n? " );
         scanf( "%d", &request );
      }

      printf( "End of run.\n" );
      fclose( cfPtr );
   }

   return 0;
}
```

Outline

3.1 Close file

```
Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run
? 1

Accounts with zero balances:
300       White           0.00

? 2

Accounts with credit balances:
400       Stone         -42.16

? 3

Accounts with debit balances:
100       Jones          24.98
200       Doe           345.67
500       Rich          224.62
? 4
End of run.
```

Outline

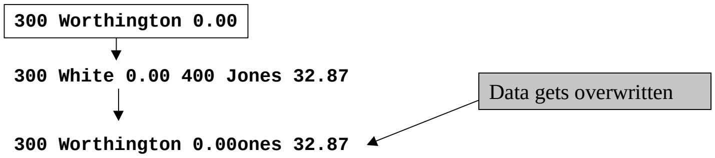
Program Output

11.5 Reading Data from a Sequential Access File (II)

- Sequential access file
  - Cannot be modified without the risk of destroying other data

```
300 White 0.00 400 Jones 32.87   (old data in file)
```

If we want to change White’s name to Worthington,



```
300 Worthington 0.00

300 White 0.00 400 Jones 32.87

300 Worthington 0.00ones 32.87
```

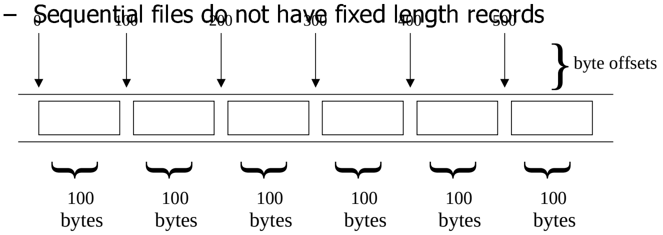
Data gets overwritten

11.5 Reading Data from a Sequential Access File (III)

- Formatted output
  - Different representation in files and screen than internal representation
  - **`1`**, **`34`**, **`-890`** are all **`int`**s, but have different sizes on disk

11.6 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



11.7 Creating a Random Access File

- Data
  - Data unformatted (stored as "raw bytes") in random access files
    - All data of the same type (`ints`, for example) use the same memory
    - All records of the same type have a fixed length
    - Data not human readable

11.7 Creating a Random Access File (II)

- 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
  - **`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
    - **`fread`** similar

11.7 Creating a Random Access File (III)

- Writing **`struct`**s

```
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

```c
/* Fig. 11.11: fig11_11.c
   Creating a randomly accessed file sequentially */
#include <stdio.h>

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

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

   if ( ( cfPtr = fopen( "credit.dat", "w" ) ) == NULL )
      printf( "File could not be opened.\n" );
   else {

      for ( i = 1; i <= 100; i++ )
         fwrite( &blankClient,
                sizeof( struct clientData ), 1, cfPtr );

      fclose( cfPtr );
   }

   return 0;
}
```

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

11.8 Writing Data Randomly to a Random Access File

- **`fseek`**
  - Sets file position pointer to a specific position
  - **`fseek( myPtr, offset, symbolic_constant);`**
    - **`myPtr`** - 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

```
/* Fig. 11.12: fig11_12.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

1. Define struct

1.1 Initialize variables

2. Open file

2.1 Input data

2.2 Write to file

```
         scanf( "%d", &client.acctNum );
      }

      fclose( cfPtr );
   }

   return 0;
}
```

```
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
```

Outline

3. Close file

Program Output

```
Enter account number
? 88
Enter lastname, firstname, balance
? Smith Dave 258.34
Enter account number
? 33
Enter lastname, firstname, balance
? Dunn Stacey 314.33
Enter account number
? 0
```

Outline

Program Output

11.9 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

```c
/* Fig. 11.15: fig11_15.c
   Reading a random access file sequentially */
#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( "%-6s%-16s%-11s%10s\n", "Acct", "Last Name",
             "First Name", "Balance" );

      while ( !feof( cfPtr ) ) {
         fread( &client, sizeof( struct clientData ), 1,
                cfPtr );

         if ( client.acctNum != 0 )
            printf( "%-6d%-16s%-11s%10.2f\n",
                   client.acctNum, client.lastName,
                   client.firstName, client.balance );
      }

```

Outline

1. Define struct

1.1 Initialize variables

2. Read (`fread`)

2.1 Print

```c
      fclose( cfPtr );
   }

   return 0;
}
```

Outline

3. Close file

Program Output

```
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
```

11.10 Example: A Transaction Processing Program

- Uses random access files to achieve instant access processing of a bank's account information

- We will
  - Update existing accounts
  - Add new accounts
  - Delete accounts
  - Store a formatted listing of all accounts in a text file

```c
/* Fig. 11.16: fig11_16.c
   This program reads a random access file sequentially,
   updates data already written to the file, creates new
   data to be placed in the file, and deletes data
   already in the file.                                  */
#include <stdio.h>

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

int enterChoice( void );
void textFile( FILE * );
void updateRecord( FILE * );
void newRecord( FILE * );
void deleteRecord( FILE * );

int main()
{
   FILE *cfPtr;
   int choice;

   if ( ( cfPtr = fopen( "credit.dat", "r+" ) ) == NULL )
      printf( "File could not be opened.\n" );
   else {

      while ( ( choice = enterChoice() ) != 5 ) {

         switch ( choice ) {
```

Outline

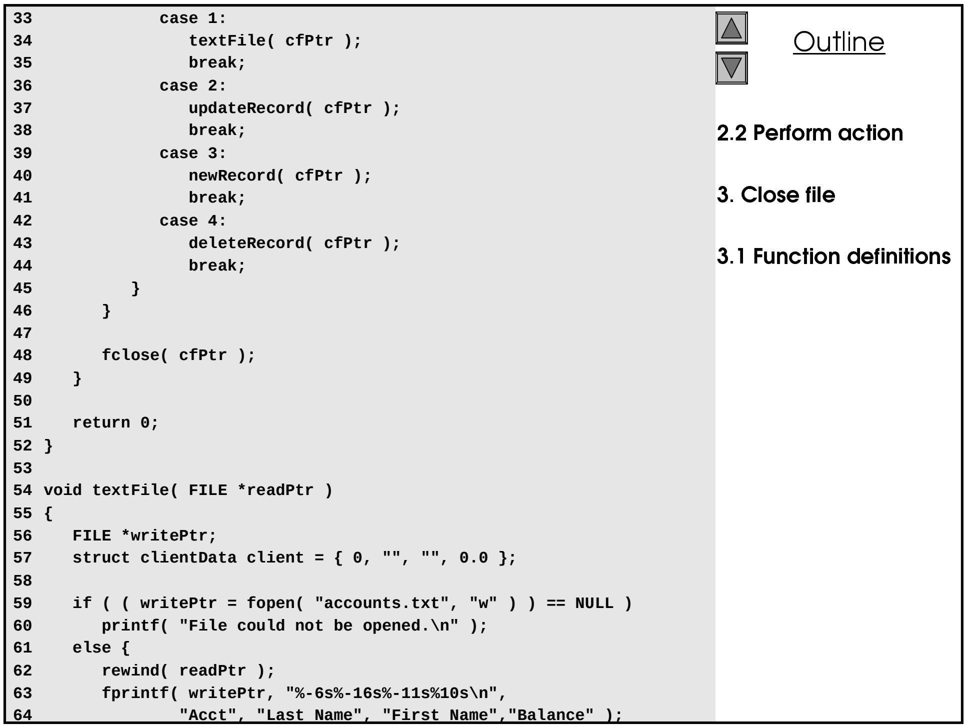
1. Define struct

1.1 Function prototypes

1.2 Initialize variables

1.3 Link pointer and open file

2. Input choice



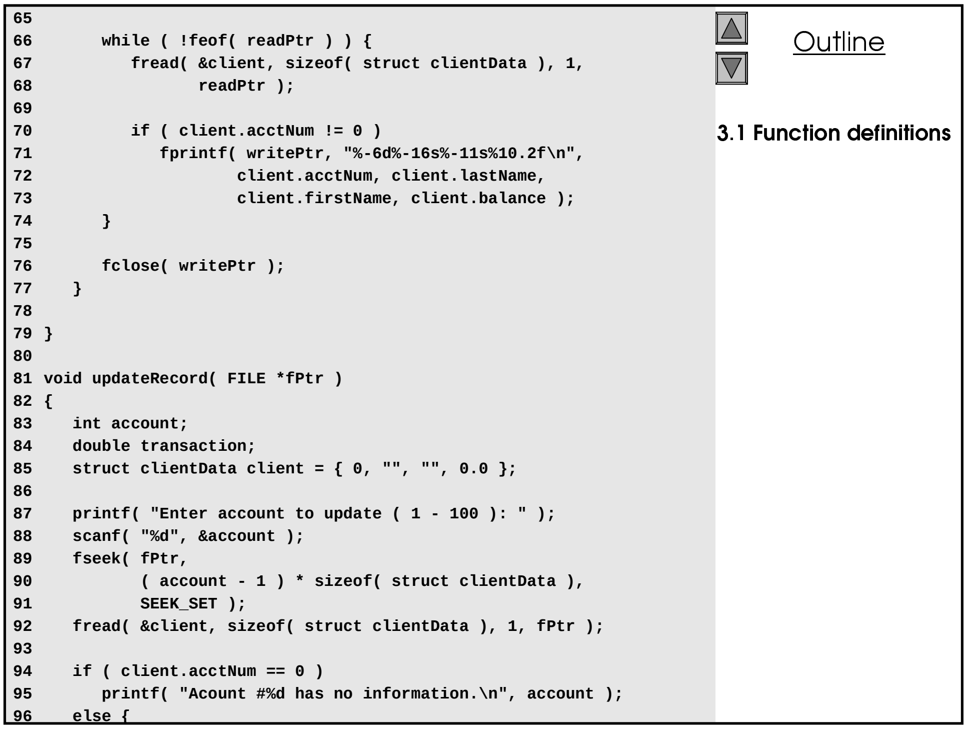
```
            case 1:
               textFile( cfPtr );
               break;
            case 2:
               updateRecord( cfPtr );
               break;
            case 3:
               newRecord( cfPtr );
               break;
            case 4:
               deleteRecord( cfPtr );
               break;
         }
      }

      fclose( cfPtr );
   }

   return 0;
}

void textFile( FILE *readPtr )
{
   FILE *writePtr;
   struct clientData client = { 0, "", "", 0.0 };

   if ( ( writePtr = fopen( "accounts.txt", "w" ) ) == NULL )
      printf( "File could not be opened.\n" );
   else {
      rewind( readPtr );
      fprintf( writePtr, "%-6s%-16s%-11s%10s\n",
              "Acct", "Last Name", "First Name","Balance" );
```

Outline

2.2 Perform action

3. Close file

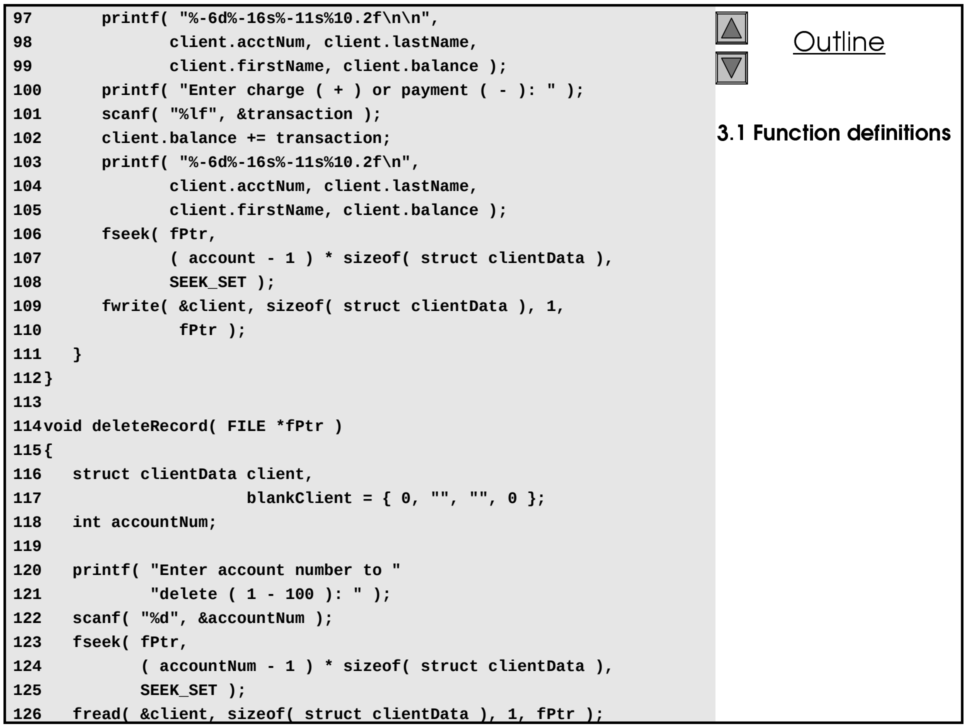
3.1 Function definitions



```

      while ( !feof( readPtr ) ) {
         fread( &client, sizeof( struct clientData ), 1,
                readPtr );

         if ( client.acctNum != 0 )
            fprintf( writePtr, "%-6d%-16s%-11s%10.2f\n",
                    client.acctNum, client.lastName,
                    client.firstName, client.balance );
      }

      fclose( writePtr );
   }

}

void updateRecord( FILE *fPtr )
{
   int account;
   double transaction;
   struct clientData client = { 0, "", "", 0.0 };

   printf( "Enter account to update ( 1 - 100 ): " );
   scanf( "%d", &account );
   fseek( fPtr,
          ( account - 1 ) * sizeof( struct clientData ),
          SEEK_SET );
   fread( &client, sizeof( struct clientData ), 1, fPtr );

   if ( client.acctNum == 0 )
      printf( "Acount #%d has no information.\n", account );
   else {
```

Outline

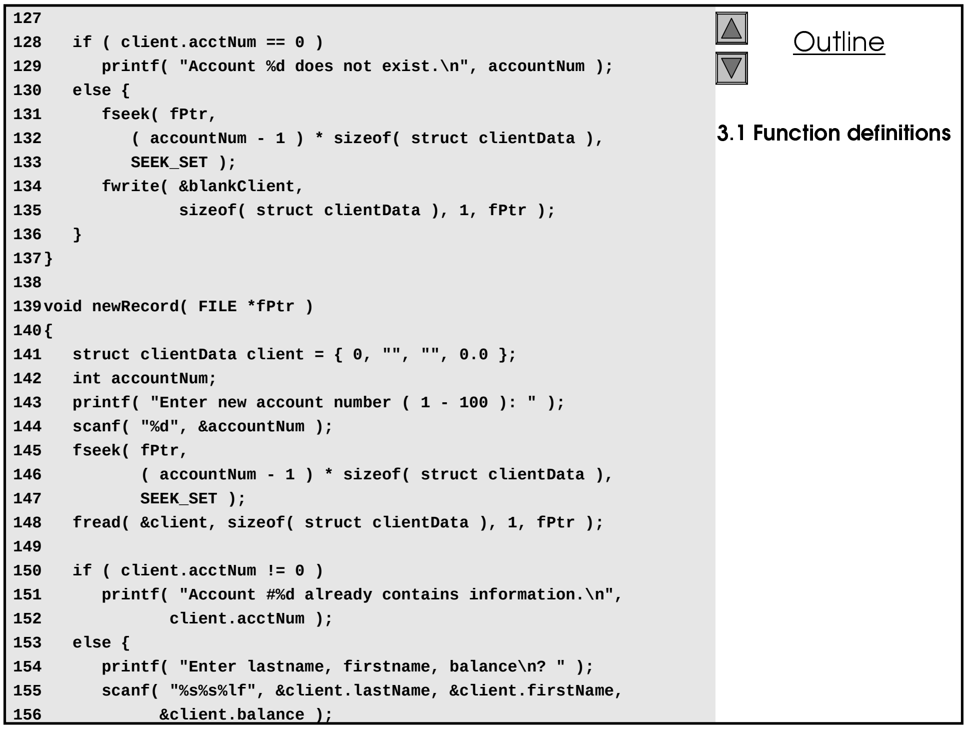
3.1 Function definitions



Outline

3.1 Function definitions

```
      printf( "%-6d%-16s%-11s%10.2f\n\n",
             client.acctNum, client.lastName,
             client.firstName, client.balance );
      printf( "Enter charge ( + ) or payment ( - ): " );
      scanf( "%lf", &transaction );
      client.balance += transaction;
      printf( "%-6d%-16s%-11s%10.2f\n",
             client.acctNum, client.lastName,
             client.firstName, client.balance );
      fseek( fPtr,
             ( account - 1 ) * sizeof( struct clientData ),
             SEEK_SET );
      fwrite( &client, sizeof( struct clientData ), 1,
              fPtr );
   }
}

void deleteRecord( FILE *fPtr )
{
   struct clientData client,
                     blankClient = { 0, "", "", 0 };
   int accountNum;

   printf( "Enter account number to "
           "delete ( 1 - 100 ): " );
   scanf( "%d", &accountNum );
   fseek( fPtr,
          ( accountNum - 1 ) * sizeof( struct clientData ),
          SEEK_SET );
   fread( &client, sizeof( struct clientData ), 1, fPtr );
```



Outline

3.1 Function definitions

```

   if ( client.acctNum == 0 )
      printf( "Account %d does not exist.\n", accountNum );
   else {
      fseek( fPtr,
         ( accountNum - 1 ) * sizeof( struct clientData ),
         SEEK_SET );
      fwrite( &blankClient,
              sizeof( struct clientData ), 1, fPtr );
   }
}

void newRecord( FILE *fPtr )
{
   struct clientData client = { 0, "", "", 0.0 };
   int accountNum;
   printf( "Enter new account number ( 1 - 100 ): " );
   scanf( "%d", &accountNum );
   fseek( fPtr,
          ( accountNum - 1 ) * sizeof( struct clientData ),
          SEEK_SET );
   fread( &client, sizeof( struct clientData ), 1, fPtr );

   if ( client.acctNum != 0 )
      printf( "Account #%d already contains information.\n",
             client.acctNum );
   else {
      printf( "Enter lastname, firstname, balance\n? " );
      scanf( "%s%s%lf", &client.lastName, &client.firstName,
            &client.balance );
      client.acctNum = accountNum;
      fseek( fPtr, ( client.acctNum - 1 ) *
            sizeof( struct clientData ), SEEK_SET );
      fwrite( &client,
              sizeof( struct clientData ), 1, fPtr );
   }
}

int enterChoice( void )
{
   int menuChoice;

   printf( "\nEnter your choice\n"
      "1 - store a formatted text file of acounts called\n"
      "    \"accounts.txt\" for printing\n"
      "2 - update an account\n"
      "3 - add a new account\n"
      "4 - delete an account\n"
      "5 - end program\n? " );
   scanf( "%d", &menuChoice );
   return menuChoice;
}
```

Outline

3.1 Function definitions

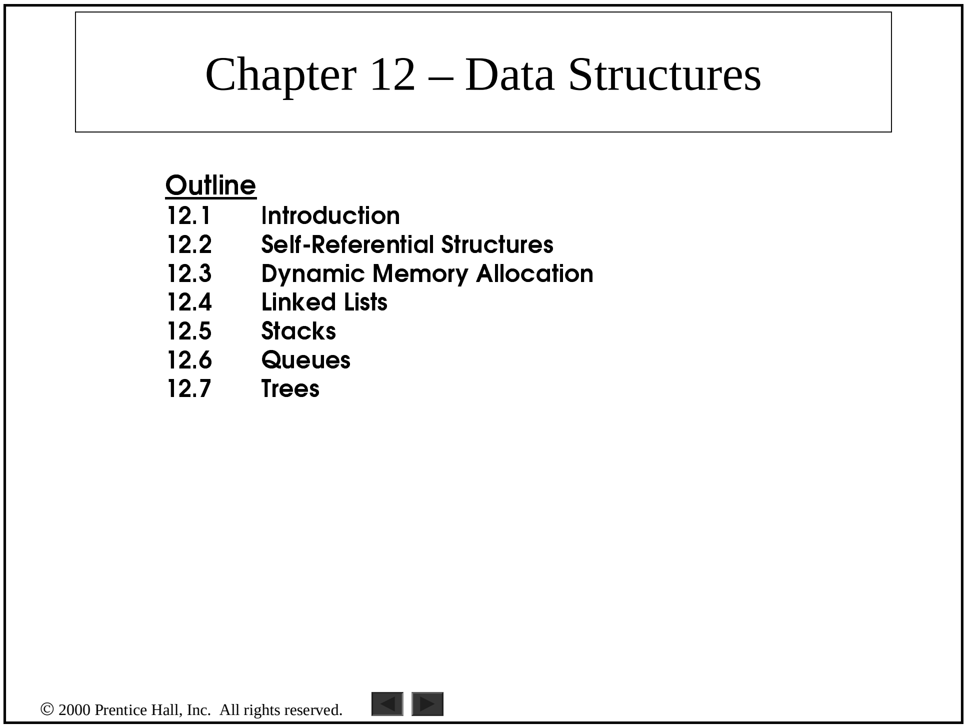
```
After choosing option 1 accounts.txt contains:

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
```

```
Enter account to update (1 - 100): 37
37    Barker          Doug             0.00

Enter charge (+) or payment (-): +87.99
37    Barker          Doug            87.99
```

```
Enter new account number (1 - 100): 22
Enter lastname, firstname, balance
? Johnston Sarah 247.45
```

Outline

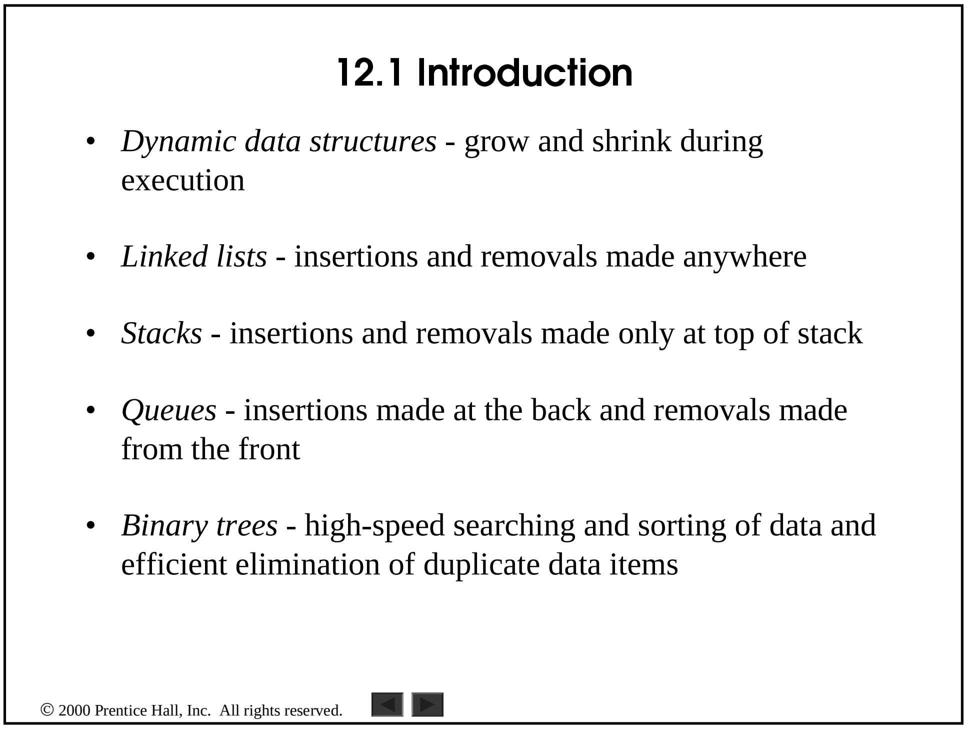
Program Output



Chapter 12 – Data Structures

Outline

12.1 Introduction
12.2 Self-Referential Structures
12.3 Dynamic Memory Allocation
12.4 Linked Lists
12.5 Stacks
12.6 Queues
12.7 Trees



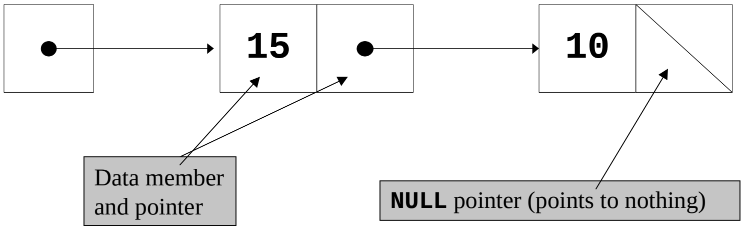
12.1 Introduction

- *Dynamic data structures* - grow and shrink during execution
- *Linked lists* - insertions and removals made anywhere
- *Stacks* - insertions and removals made only at top of stack
- *Queues* - insertions made at the back and removals made from the front
- *Binary trees* - high-speed searching and sorting of data and efficient elimination of duplicate data items

12.2 Self-Referential Structures

- Self-referential structures
  - Structure that contains a pointer to a structure of the same type
  - Can be linked together to form useful data structures such as lists, queues, stacks and trees
  - Terminated with a **`NULL`** pointer (**`0`**)

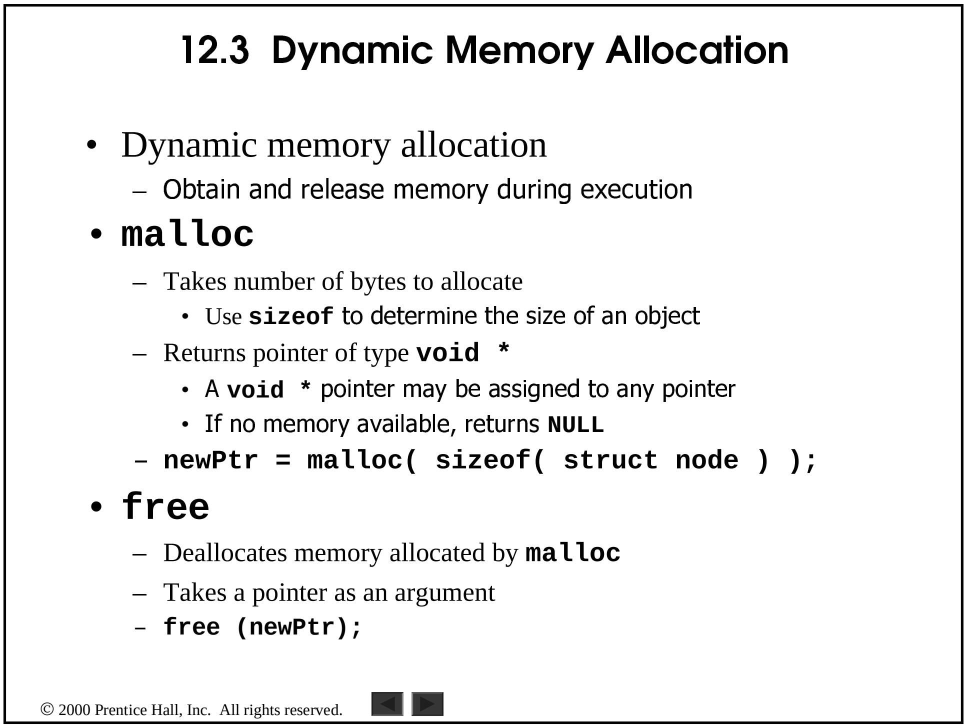
- Two self-referential structure objects linked together



12.2 Self-Referential Classes (II)

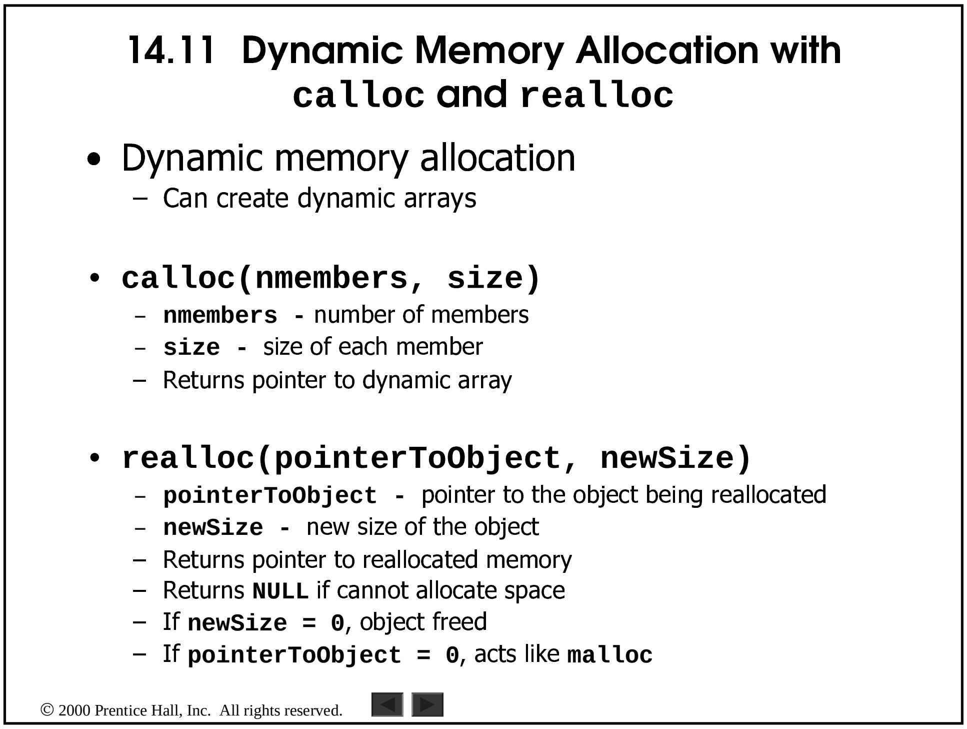
```
struct node {
   int data;
   struct node *nextPtr;
}
```

- **`nextPtr`** - points to an object of type **`node`**
  - Referred to as a *link* – ties one **`node`** to another **`node`**



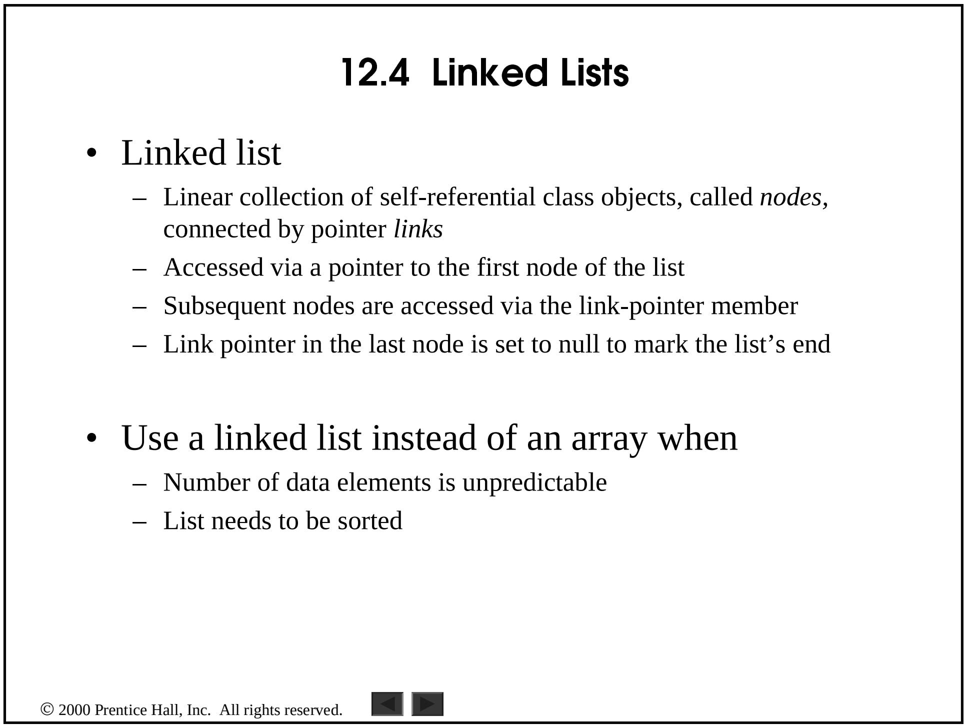
12.3 Dynamic Memory Allocation

- Dynamic memory allocation
  - Obtain and release memory during execution
- **`malloc`**
  - Takes number of bytes to allocate
    - Use **`sizeof`** to determine the size of an object
  - Returns pointer of type **`void *`**
    - A **`void *`** pointer may be assigned to any pointer
    - If no memory available, returns **`NULL`**
  - **`newPtr = malloc( sizeof( struct node ) );`**
- **`free`**
  - Deallocates memory allocated by **`malloc`**
  - Takes a pointer as an argument
  - **`free (newPtr);`**



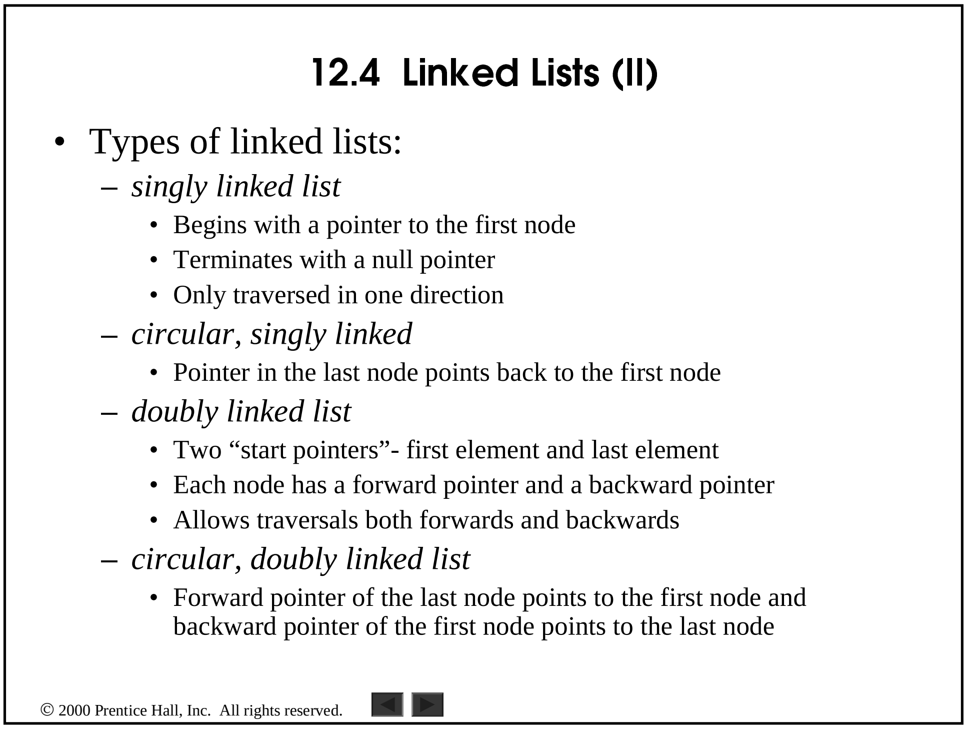
14.11 Dynamic Memory Allocation with `calloc` and `realloc`

- Dynamic memory allocation
  - Can create dynamic arrays
- **`calloc(nmembers, size)`**
  - **`nmembers -`** number of members
  - **`size -`** size of each member
  - Returns pointer to dynamic array
- **`realloc(pointerToObject, newSize)`**
  - **`pointerToObject -`** pointer to the object being reallocated
  - **`newSize -`** new size of the object
  - Returns pointer to reallocated memory
  - Returns **`NULL`** if cannot allocate space
  - If **`newSize = 0`**, object freed
  - If **`pointerToObject = 0`**, acts like **`malloc`**



12.4 Linked Lists

- Linked list
  - Linear collection of self-referential class objects, called *nodes*, connected by pointer *links*
  - Accessed via a pointer to the first node of the list
  - Subsequent nodes are accessed via the link-pointer member
  - Link pointer in the last node is set to null to mark the list's end
- Use a linked list instead of an array when
  - Number of data elements is unpredictable
  - List needs to be sorted



12.4 Linked Lists (II)

- Types of linked lists:
  - *singly linked list*
    - Begins with a pointer to the first node
    - Terminates with a null pointer
    - Only traversed in one direction
  - *circular, singly linked*
    - Pointer in the last node points back to the first node
  - *doubly linked list*
    - Two "start pointers"- first element and last element
    - Each node has a forward pointer and a backward pointer
    - Allows traversals both forwards and backwards
  - *circular, doubly linked list*
    - Forward pointer of the last node points to the first node and backward pointer of the first node points to the last node

```c
/* Fig. 12.3: fig12_03.c
   Operating and maintaining a list */
#include <stdio.h>
#include <stdlib.h>

struct listNode {   /* self-referential structure */
   char data;
   struct listNode *nextPtr;
};

typedef struct listNode ListNode;
typedef ListNode *ListNodePtr;

void insert( ListNodePtr *, char );
char delete( ListNodePtr *, char );
int isEmpty( ListNodePtr );
void printList( ListNodePtr );
void instructions( void );

int main()
{
   ListNodePtr startPtr = NULL;
   int choice;
   char item;

   instructions();  /* display the menu */
   printf( "? " );
   scanf( "%d", &choice );
```

Outline

1. Define struct

1.1 Function prototypes

1.2 Initialize variables

2. Input choice

```c

   while ( choice != 3 ) {

      switch ( choice ) {
         case 1:
            printf( "Enter a character: " );
            scanf( "\n%c", &item );
            insert( &startPtr, item );
            printList( startPtr );
            break;
         case 2:
            if ( !isEmpty( startPtr ) ) {
               printf( "Enter character to be deleted: " );
               scanf( "\n%c", &item );

               if ( delete( &startPtr, item ) ) {
                  printf( "%c deleted.\n", item );
                  printList( startPtr );
               }
               else
                  printf( "%c not found.\n\n", item );
            }
            else
               printf( "List is empty.\n\n" );

            break;
         default:
            printf( "Invalid choice.\n\n" );
            instructions();
            break;
      }
```

Outline

2.1 `switch` statement

```

      printf( "? " );
      scanf( "%d", &choice );
   }

   printf( "End of run.\n" );
   return 0;
}

/* Print the instructions */
void instructions( void )
{
   printf( "Enter your choice:\n"
          "   1 to insert an element into the list.\n"
          "   2 to delete an element from the list.\n"
          "   3 to end.\n" );
}

/* Insert a new value into the list in sorted order */
void insert( ListNodePtr *sPtr, char value )
{
   ListNodePtr newPtr, previousPtr, currentPtr;

   newPtr = malloc( sizeof( ListNode ) );

   if ( newPtr != NULL ) {     /* is space available */
      newPtr->data = value;
      newPtr->nextPtr = NULL;

      previousPtr = NULL;
      currentPtr = *sPtr;
```

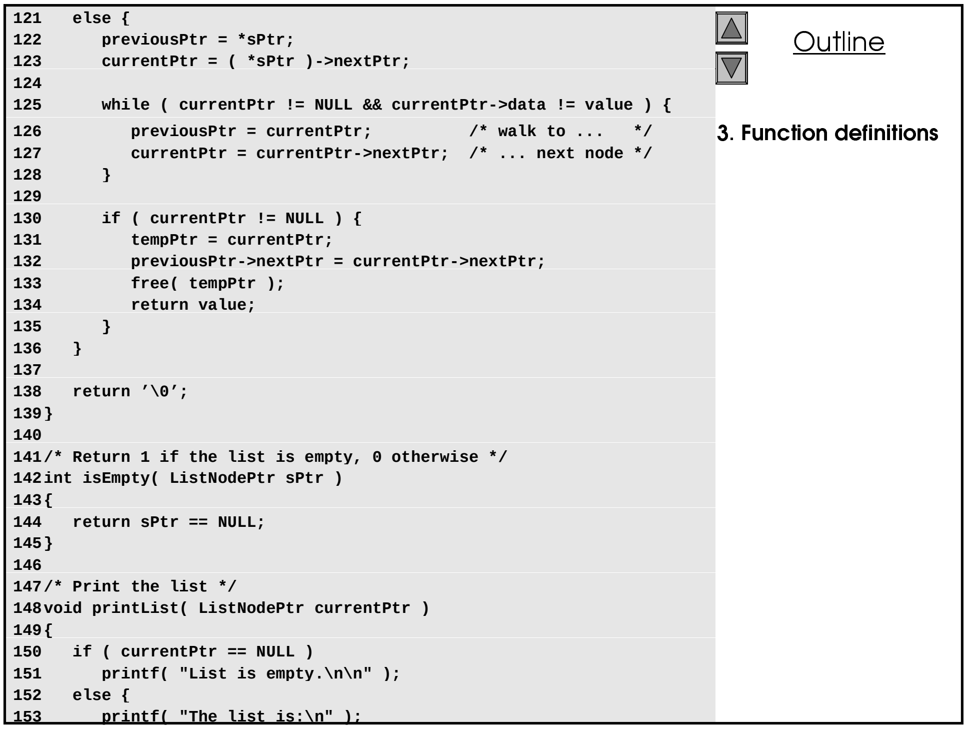
Outline

3. Function definitions

```

      while ( currentPtr != NULL && value > currentPtr->data ) {
         previousPtr = currentPtr;          /* walk to ...   */
         currentPtr = currentPtr->nextPtr;  /* ... next node */
      }

      if ( previousPtr == NULL ) {
         newPtr->nextPtr = *sPtr;
         *sPtr = newPtr;
      }
      else {
         previousPtr->nextPtr = newPtr;
         newPtr->nextPtr = currentPtr;
      }
   }
   else
      printf( "%c not inserted. No memory available.\n", value );
}

/* Delete a list element */
char delete( ListNodePtr *sPtr, char value )
{
   ListNodePtr previousPtr, currentPtr, tempPtr;

   if ( value == ( *sPtr )->data ) {
      tempPtr = *sPtr;
      *sPtr = ( *sPtr )->nextPtr;  /* de-thread the node */
      free( tempPtr );             /* free the de-threaded node */
      return value;
   }
```

Outline

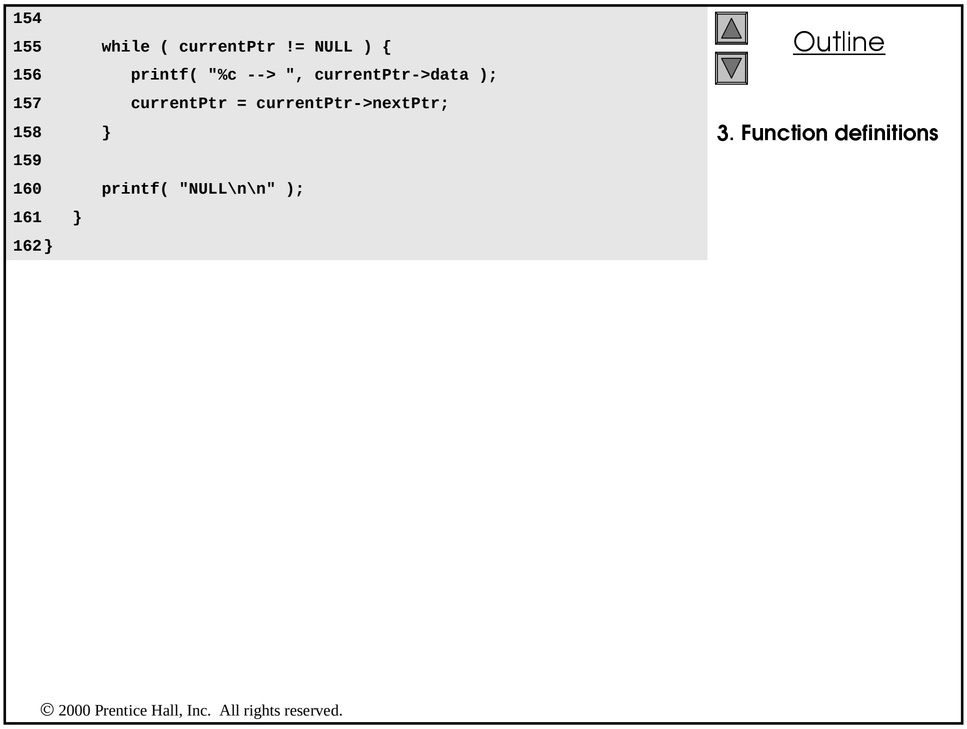
3. Function definitions



Outline

3. Function definitions

```
   else {
      previousPtr = *sPtr;
      currentPtr = ( *sPtr )->nextPtr;

      while ( currentPtr != NULL && currentPtr->data != value ) {
         previousPtr = currentPtr;          /* walk to ...   */
         currentPtr = currentPtr->nextPtr;  /* ... next node */
      }

      if ( currentPtr != NULL ) {
         tempPtr = currentPtr;
         previousPtr->nextPtr = currentPtr->nextPtr;
         free( tempPtr );
         return value;
      }
   }

   return '\0';
}

/* Return 1 if the list is empty, 0 otherwise */
int isEmpty( ListNodePtr sPtr )
{
   return sPtr == NULL;
}

/* Print the list */
void printList( ListNodePtr currentPtr )
{
   if ( currentPtr == NULL )
      printf( "List is empty.\n\n" );
   else {
      printf( "The list is:\n" );
```



Outline

3. Function definitions

```

      while ( currentPtr != NULL ) {
         printf( "%c --> ", currentPtr->data );
         currentPtr = currentPtr->nextPtr;
      }

      printf( "NULL\n\n" );
   }
}
```

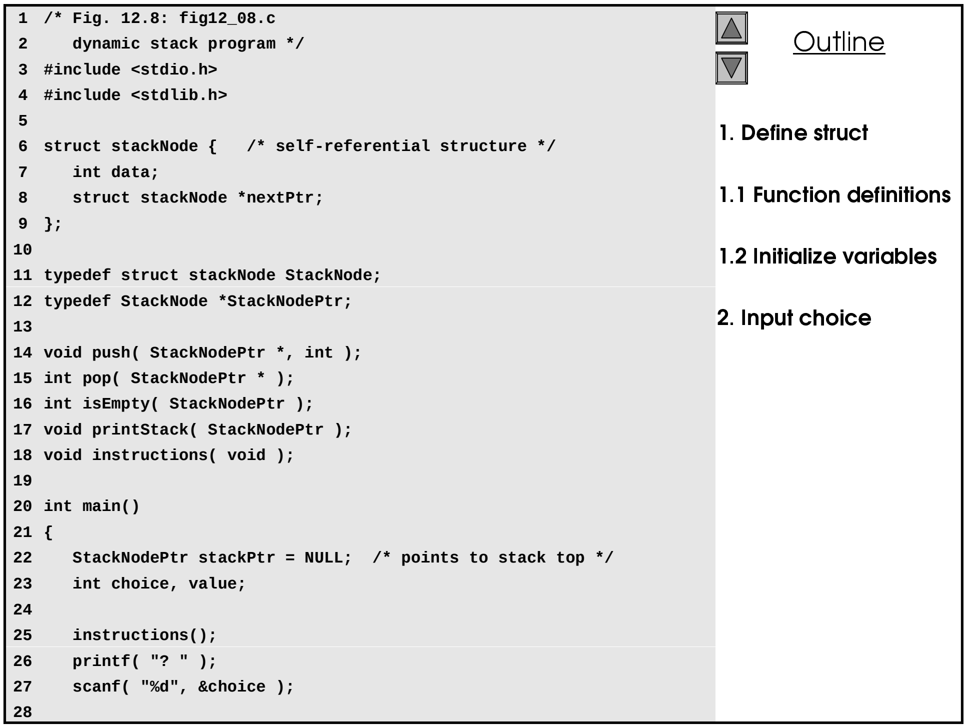
```
Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 1
Enter a character: B
The list is:
B --> NULL

? 1
Enter a character: A
The list is:
A --> B --> NULL

? 1
Enter a character: C
The list is:
A --> B --> C --> NULL

? 2
Enter character to be deleted: D
D not found.

? 2
Enter character to be deleted: B
B deleted.
The list is:
A --> C --> NULL
```

Outline

Program Output

12.5 Stacks

- Stack
  - New nodes can be added and removed only at the top
  - Similar to a pile of dishes
  - Last-in, first-out (LIFO)
  - Bottom of stack indicated by a link member to **null**
  - Constrained version of a linked list
- push
  - Adds a new node to the top of the stack
- pop
  - Removes a node from the top
  - Stores the popped value
  - Returns **true** if **pop** was successful



```c
/* Fig. 12.8: fig12_08.c
   dynamic stack program */
#include <stdio.h>
#include <stdlib.h>

struct stackNode {   /* self-referential structure */
   int data;
   struct stackNode *nextPtr;
};

typedef struct stackNode StackNode;
typedef StackNode *StackNodePtr;

void push( StackNodePtr *, int );
int pop( StackNodePtr * );
int isEmpty( StackNodePtr );
void printStack( StackNodePtr );
void instructions( void );

int main()
{
   StackNodePtr stackPtr = NULL;  /* points to stack top */
   int choice, value;

   instructions();
   printf( "? " );
   scanf( "%d", &choice );

```

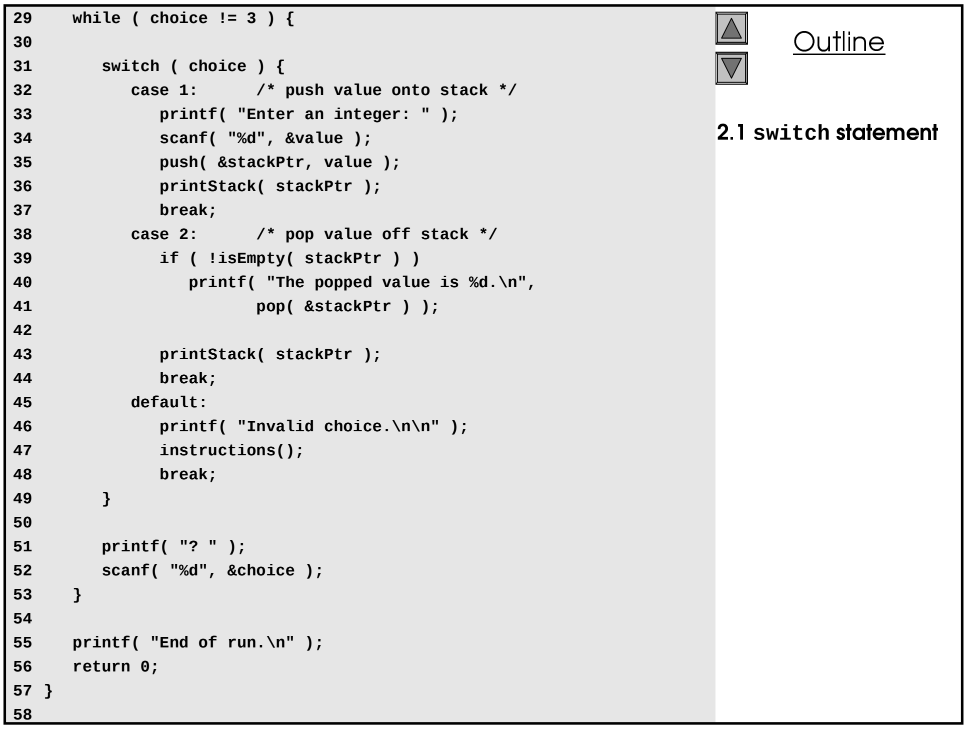
Outline

1. Define struct

1.1 Function definitions

1.2 Initialize variables

2. Input choice



```c
   while ( choice != 3 ) {

      switch ( choice ) {
         case 1:      /* push value onto stack */
            printf( "Enter an integer: " );
            scanf( "%d", &value );
            push( &stackPtr, value );
            printStack( stackPtr );
            break;
         case 2:      /* pop value off stack */
            if ( !isEmpty( stackPtr ) )
               printf( "The popped value is %d.\n",
                      pop( &stackPtr ) );

            printStack( stackPtr );
            break;
         default:
            printf( "Invalid choice.\n\n" );
            instructions();
            break;
      }

      printf( "? " );
      scanf( "%d", &choice );
   }

   printf( "End of run.\n" );
   return 0;
}

```

Outline

2.1 switch statement

```
/* Print the instructions */
void instructions( void )
{
   printf( "Enter choice:\n"
          "1 to push a value on the stack\n"
          "2 to pop a value off the stack\n"
          "3 to end program\n" );
}

/* Insert a node at the stack top */
void push( StackNodePtr *topPtr, int info )
{
   StackNodePtr newPtr;

   newPtr = malloc( sizeof( StackNode ) );
   if ( newPtr != NULL ) {
      newPtr->data = info;
      newPtr->nextPtr = *topPtr;
      *topPtr = newPtr;
   }
   else
      printf( "%d not inserted. No memory available.\n",
              info );
}

```

Outline

3. Function definitions

```
/* Remove a node from the stack top */
int pop( StackNodePtr *topPtr )
{
   StackNodePtr tempPtr;
   int popValue;

   tempPtr = *topPtr;
   popValue = ( *topPtr )->data;
   *topPtr = ( *topPtr )->nextPtr;
   free( tempPtr );
   return popValue;
}

/* Print the stack */
void printStack( StackNodePtr currentPtr )
{
   if ( currentPtr == NULL )
      printf( "The stack is empty.\n\n" );
   else {
      printf( "The stack is:\n" );

      while ( currentPtr != NULL ) {
         printf( "%d --> ", currentPtr->data );
         currentPtr = currentPtr->nextPtr;
      }

      printf( "NULL\n\n" );
   }
}

```

Outline

3. Function definitions

```
/* Is the stack empty? */
int isEmpty( StackNodePtr topPtr )
{
   return topPtr == NULL;
}
```

```
Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 1
Enter an integer: 5
The stack is:
5 --> NULL

? 1
Enter an integer: 6
The stack is:
6 --> 5 --> NULL

? 1
Enter an integer: 4
The stack is:
4 --> 6 --> 5 --> NULL

? 2
The popped value is 4.
The stack is:
6 --> 5 --> NULL
```

Outline

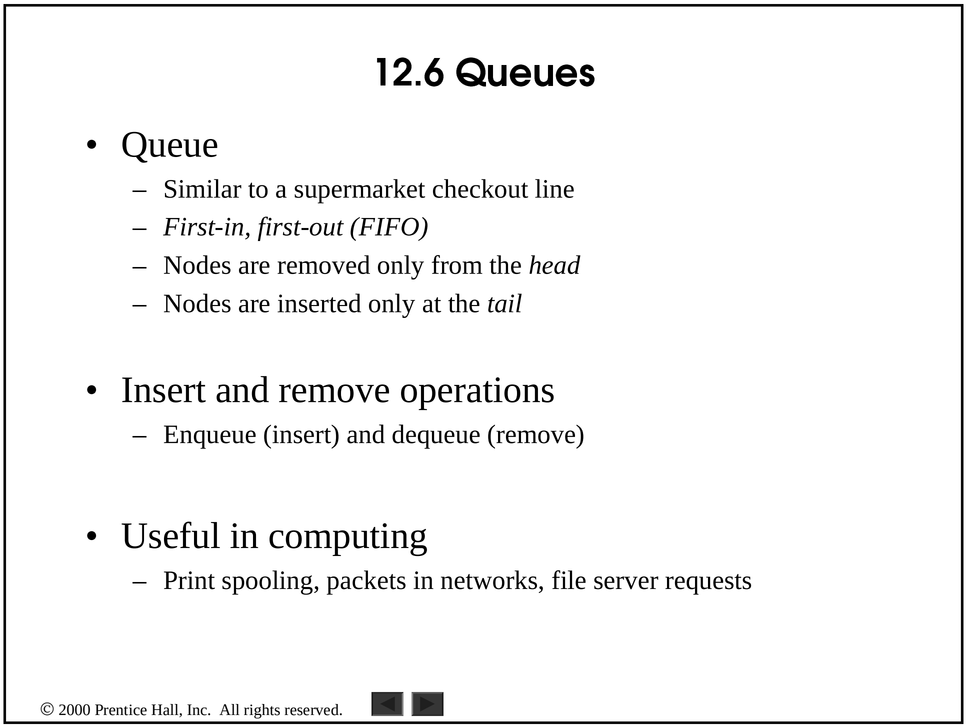
3. Function definitions

Program Output

```
? 2
The popped value is 6.
The stack is:
5 --> NULL

? 2
The popped value is 5.
The stack is empty.

? 2
The stack is empty.

? 4
Invalid choice.

Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 3
End of run.
```

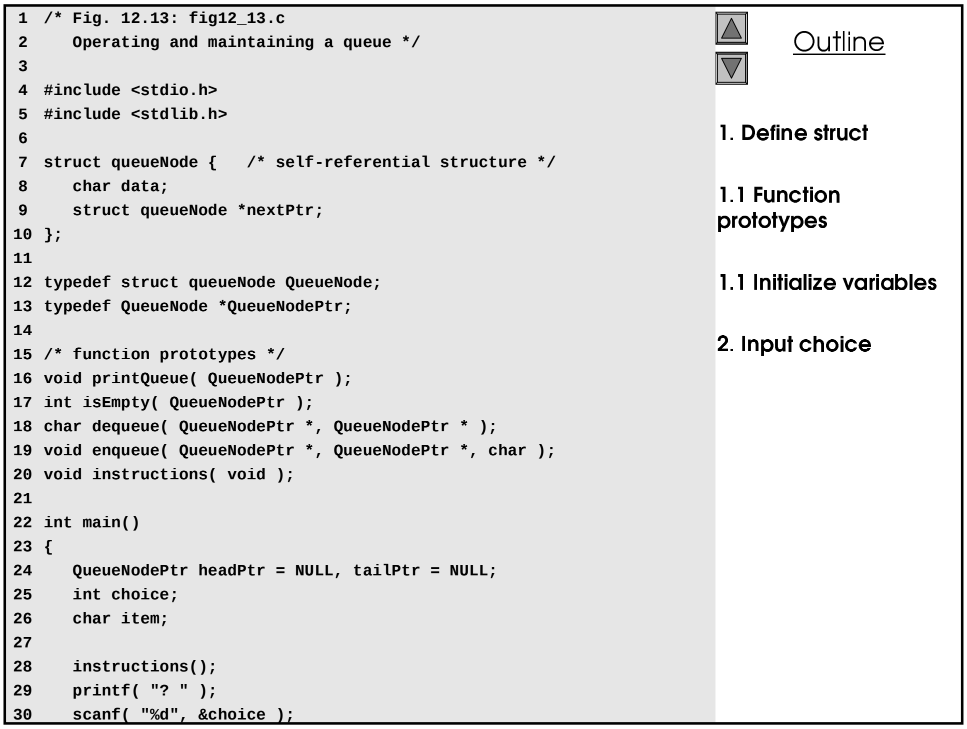
Outline

Program Output



12.6 Queues

- Queue
  - Similar to a supermarket checkout line
  - *First-in, first-out (FIFO)*
  - Nodes are removed only from the *head*
  - Nodes are inserted only at the *tail*
- Insert and remove operations
  - Enqueue (insert) and dequeue (remove)
- Useful in computing
  - Print spooling, packets in networks, file server requests



```
/* Fig. 12.13: fig12_13.c
   Operating and maintaining a queue */

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

struct queueNode {   /* self-referential structure */
   char data;
   struct queueNode *nextPtr;
};

typedef struct queueNode QueueNode;
typedef QueueNode *QueueNodePtr;

/* function prototypes */
void printQueue( QueueNodePtr );
int isEmpty( QueueNodePtr );
char dequeue( QueueNodePtr *, QueueNodePtr * );
void enqueue( QueueNodePtr *, QueueNodePtr *, char );
void instructions( void );

int main()
{
   QueueNodePtr headPtr = NULL, tailPtr = NULL;
   int choice;
   char item;

   instructions();
   printf( "? " );
   scanf( "%d", &choice );
```

Outline

1. Define struct

1.1 Function prototypes

1.1 Initialize variables

2. Input choice

```
   while ( choice != 3 ) {

      switch( choice ) {

         case 1:
            printf( "Enter a character: " );
            scanf( "\n%c", &item );
            enqueue( &headPtr, &tailPtr, item );
            printQueue( headPtr );
            break;
         case 2:
            if ( !isEmpty( headPtr ) ) {
               item = dequeue( &headPtr, &tailPtr );
               printf( "%c has been dequeued.\n", item );
            }

            printQueue( headPtr );
            break;

         default:
            printf( "Invalid choice.\n\n" );
            instructions();
            break;
      }

      printf( "? " );
      scanf( "%d", &choice );
   }

   printf( "End of run.\n" );
   return 0;
}
```

Outline

2.1 `switch` statement

```
void instructions( void )
{
   printf ( "Enter your choice:\n"
           "   1 to add an item to the queue\n"
           "   2 to remove an item from the queue\n"
           "   3 to end\n" );
}

void enqueue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr,
              char value )
{
   QueueNodePtr newPtr;

   newPtr = malloc( sizeof( QueueNode ) );

   if ( newPtr != NULL ) {
      newPtr->data = value;
      newPtr->nextPtr = NULL;

      if ( isEmpty( *headPtr ) )
         *headPtr = newPtr;
      else
         ( *tailPtr )->nextPtr = newPtr;

      *tailPtr = newPtr;
   }
   else
      printf( "%c not inserted. No memory available.\n",
              value );
}
```

Outline

3. Function definitions

```
char dequeue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr )
{
   char value;
   QueueNodePtr tempPtr;

   value = ( *headPtr )->data;
   tempPtr = *headPtr;
   *headPtr = ( *headPtr )->nextPtr;

   if ( *headPtr == NULL )
      *tailPtr = NULL;

   free( tempPtr );
   return value;
}

int isEmpty( QueueNodePtr headPtr )
{
   return headPtr == NULL;
}

void printQueue( QueueNodePtr currentPtr )
{
   if ( currentPtr == NULL )
      printf( "Queue is empty.\n\n" );
   else {
      printf( "The queue is:\n" );
```

Outline

3. Function definitions

```

      while ( currentPtr != NULL ) {
         printf( "%c --> ", currentPtr->data );
         currentPtr = currentPtr->nextPtr;
      }

      printf( "NULL\n\n" );
   }
}
```

Outline

3. Function definitions

```
Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 1
Enter a character: A
The queue is:
A --> NULL

? 1
Enter a character: B
The queue is:
A --> B --> NULL

? 1
Enter a character: C
The queue is:
A --> B --> C --> NULL
```

Program Output

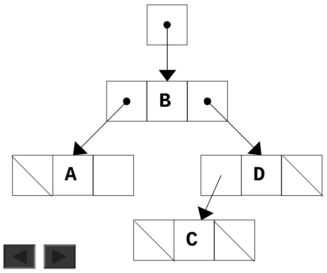
```
? 2
A has been dequeued.
The queue is:
B --> C --> NULL

? 2
B has been dequeued.
The queue is:
C --> NULL

? 2
C has been dequeued.
Queue is empty.

? 2
Queue is empty.

? 4
Invalid choice.

Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 3
End of run.
```

Outline

Program Output

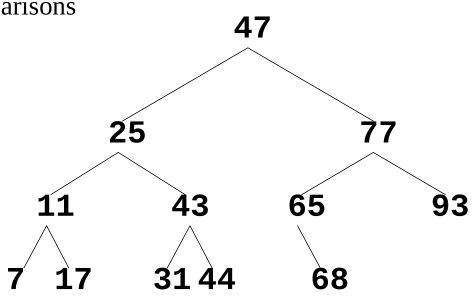
12.7 Trees

- Tree nodes contain two or more links
  - All other data structures we have discussed only contain one
- Binary trees
  - All nodes contain two links
    - None, one, or both of which may be **`NULL`**
  - The *root node* is the first node in a tree.
  - Each link in the root node refers to a *child*
  - A node with no children is called a leaf node



12.7 Trees (II)

- Binary search tree
  - Values in left subtree less than parent
  - Values in right subtree greater than parent
  - Facilitates *duplicate elimination*
  - Fast searches - for a balanced tree, maximum of $\log_2 n$ comparisons



12.7 Trees (III)

- Tree traversals:
  - Inorder traversal - prints the node values in ascending order
    1. Traverse the left subtree with an inorder traversal.
    2. Process the value in the node (i.e., print the node value).
    3. Traverse the right subtree with an inorder traversal.
  - Preorder traversal:
    1. Process the value in the node.
    2. Traverse the left subtree with a preorder traversal.
    3. Traverse the right subtree with a preorder traversal.
  - Postorder traversal:
    1. Traverse the left subtree with a postorder traversal.
    2. Traverse the right subtree with a postorder traversal.
    3. Process the value in the node

```c
/* Fig. 12.19: fig12_19.c
   Create a binary tree and traverse it
   preorder, inorder, and postorder */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct treeNode {
   struct treeNode *leftPtr;
   int data;
   struct treeNode *rightPtr;
};

typedef struct treeNode TreeNode;
typedef TreeNode *TreeNodePtr;

void insertNode( TreeNodePtr *, int );
void inOrder( TreeNodePtr );
void preOrder( TreeNodePtr );
void postOrder( TreeNodePtr );

int main()
{
   int i, item;
   TreeNodePtr rootPtr = NULL;

   srand( time( NULL ) );

```

Outline

1. Define structure

1.1 Function prototypes

1.2 Initialize variables

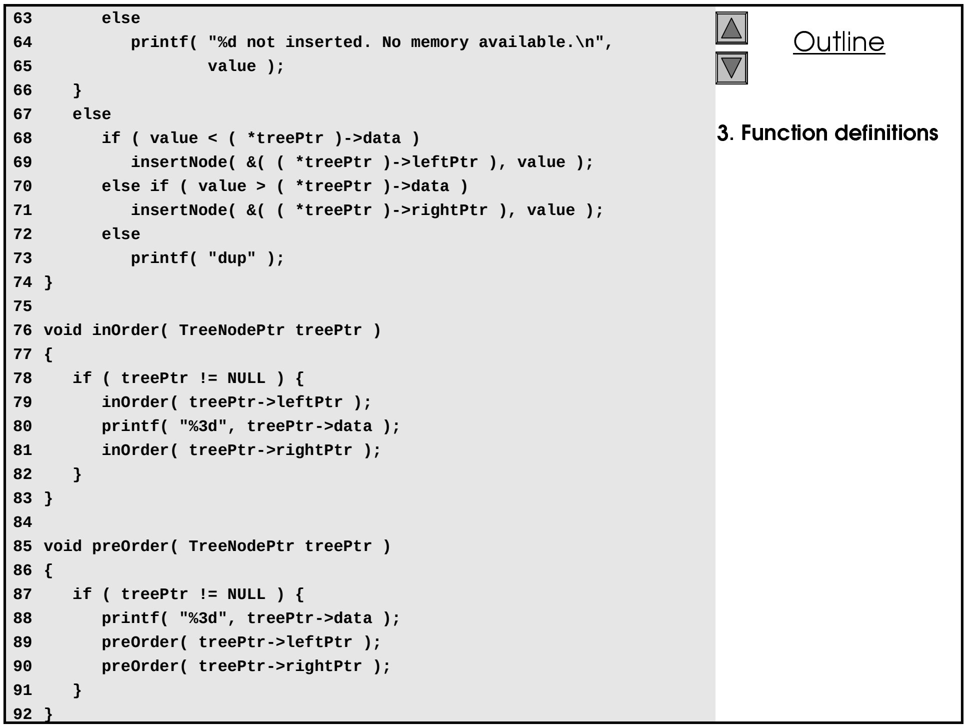
```c
   /* insert random values between 1 and 15 in the tree */
   printf( "The numbers being placed in the tree are:\n" );

   for ( i = 1; i <= 10; i++ ) {
      item = rand() % 15;
      printf( "%3d", item );
      insertNode( &rootPtr, item );
   }

   /* traverse the tree preOrder */
   printf( "\n\nThe preOrder traversal is:\n" );
   preOrder( rootPtr );

   /* traverse the tree inOrder */
   printf( "\n\nThe inOrder traversal is:\n" );
   inOrder( rootPtr );

   /* traverse the tree postOrder */
   printf( "\n\nThe postOrder traversal is:\n" );
   postOrder( rootPtr );

   return 0;
}

void insertNode( TreeNodePtr *treePtr, int value )
{
   if ( *treePtr == NULL ) {    /* *treePtr is NULL */
      *treePtr = malloc( sizeof( TreeNode ) );

      if ( *treePtr != NULL ) {
         ( *treePtr )->data = value;
         ( *treePtr )->leftPtr = NULL;
         ( *treePtr )->rightPtr = NULL;
      }
```

Outline

1.3 Insert random elements

2. Function calls

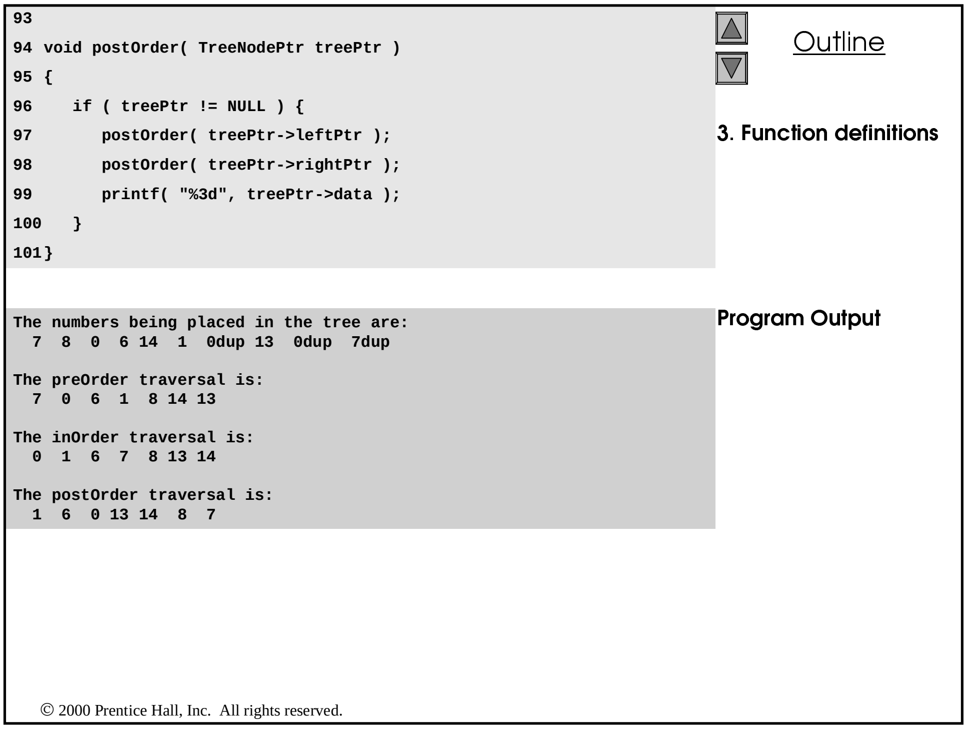
3. Function definitions



Outline

3. Function definitions

```
      else
         printf( "%d not inserted. No memory available.\n",
                 value );
   }
   else
      if ( value < ( *treePtr )->data )
         insertNode( &( ( *treePtr )->leftPtr ), value );
      else if ( value > ( *treePtr )->data )
         insertNode( &( ( *treePtr )->rightPtr ), value );
      else
         printf( "dup" );
}

void inOrder( TreeNodePtr treePtr )
{
   if ( treePtr != NULL ) {
      inOrder( treePtr->leftPtr );
      printf( "%3d", treePtr->data );
      inOrder( treePtr->rightPtr );
   }
}

void preOrder( TreeNodePtr treePtr )
{
   if ( treePtr != NULL ) {
      printf( "%3d", treePtr->data );
      preOrder( treePtr->leftPtr );
      preOrder( treePtr->rightPtr );
   }
}
```



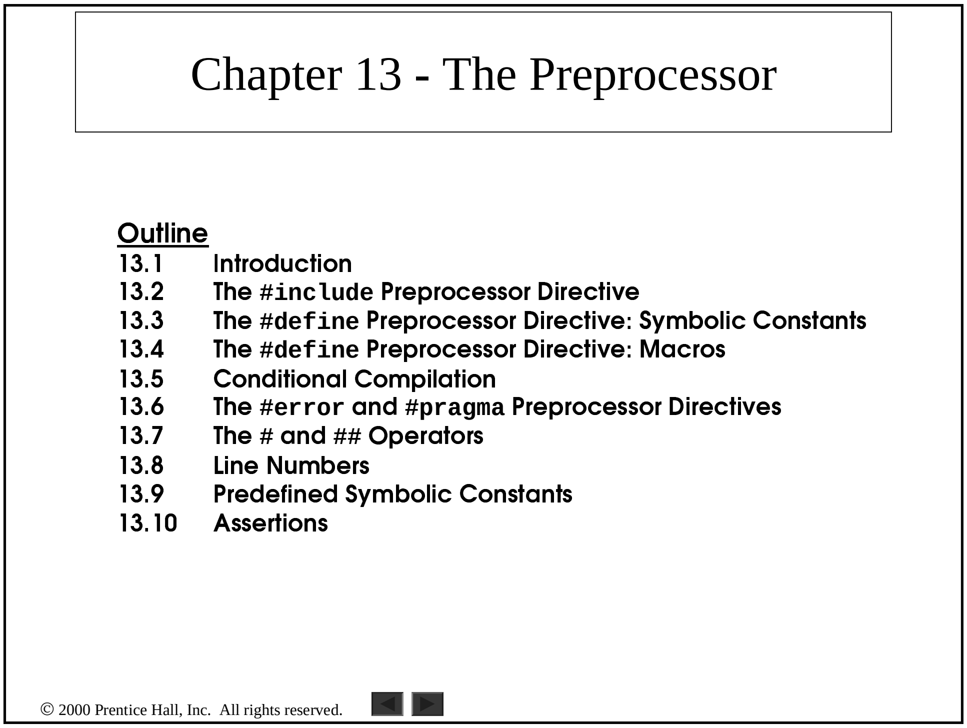
Outline

3. Function definitions

```

void postOrder( TreeNodePtr treePtr )
{
   if ( treePtr != NULL ) {
      postOrder( treePtr->leftPtr );
      postOrder( treePtr->rightPtr );
      printf( "%3d", treePtr->data );
   }
}
```

Program Output

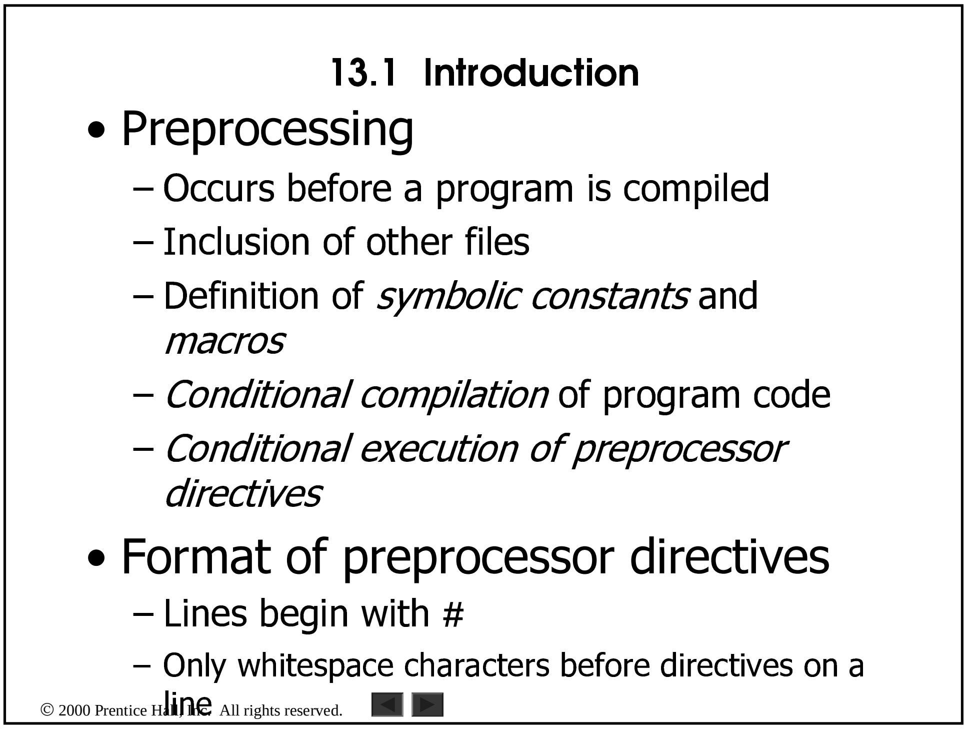
```
The numbers being placed in the tree are:
  7  8  0  6 14  1  0dup 13  0dup  7dup

The preOrder traversal is:
  7  0  6  1  8 14 13

The inOrder traversal is:
  0  1  6  7  8 13 14

The postOrder traversal is:
  1  6  0 13 14  8  7
```



Chapter 13 - The Preprocessor

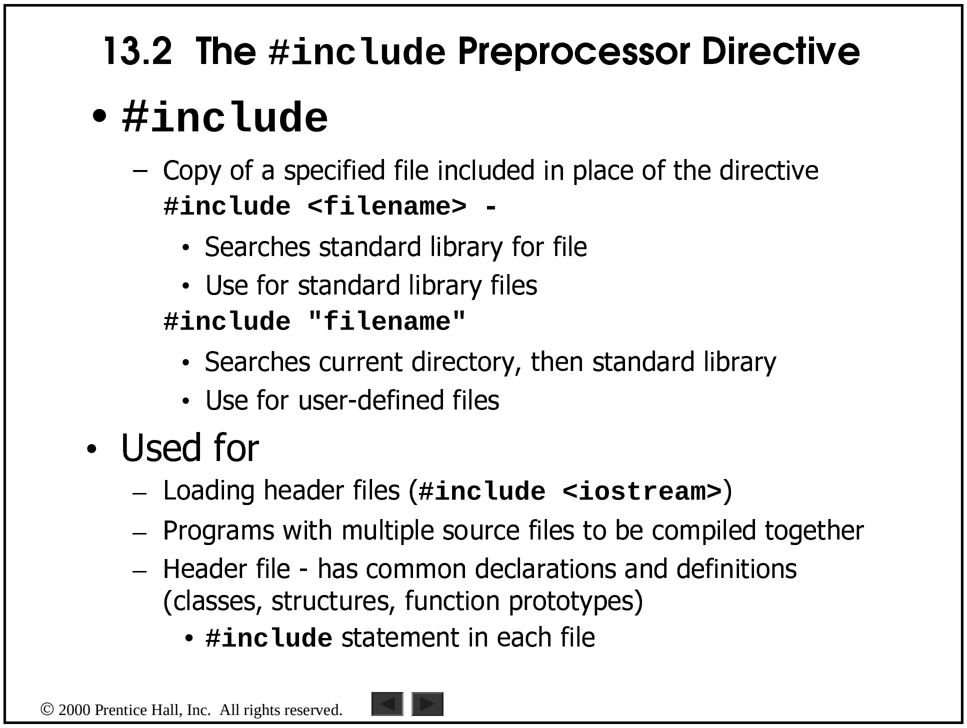
Outline

13.1 Introduction
13.2 The `#include` Preprocessor Directive
13.3 The `#define` Preprocessor Directive: Symbolic Constants
13.4 The `#define` Preprocessor Directive: Macros
13.5 Conditional Compilation
13.6 The `#error` and `#pragma` Preprocessor Directives
13.7 The # and ## Operators
13.8 Line Numbers
13.9 Predefined Symbolic Constants
13.10 Assertions



13.1 Introduction

- Preprocessing
  - – Occurs before a program is compiled
  - – Inclusion of other files
  - – Definition of *symbolic constants* and *macros*
  - – *Conditional compilation* of program code
  - – *Conditional execution of preprocessor directives*
- Format of preprocessor directives
  - – Lines begin with #
  - – Only whitespace characters before directives on a line

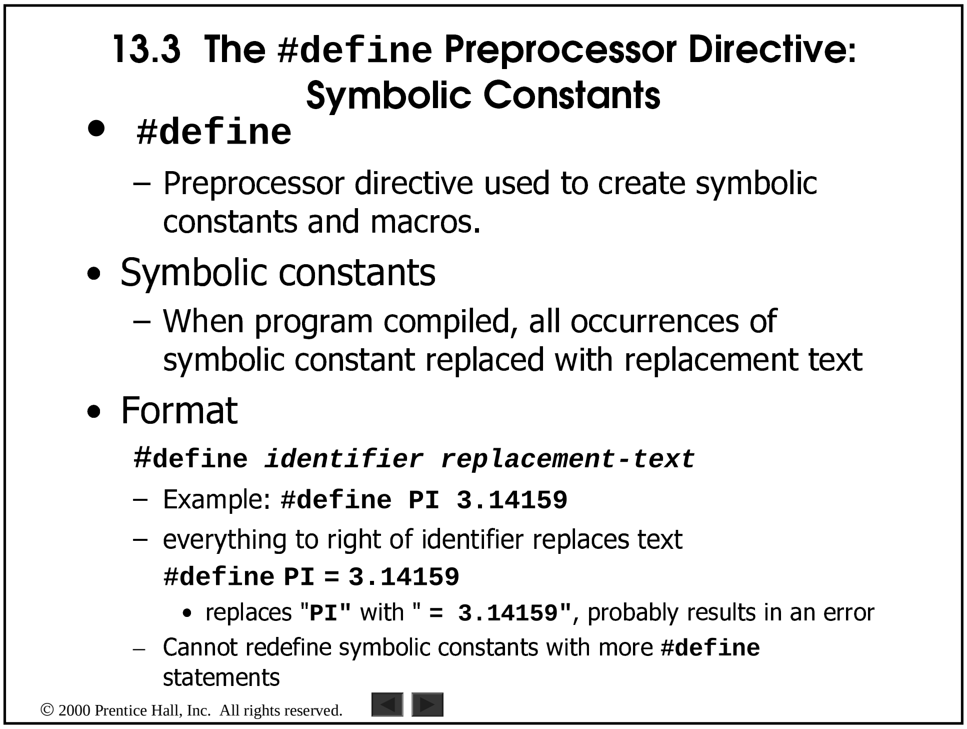


13.2 The `#include` Preprocessor Directive

- **`#include`**
  - Copy of a specified file included in place of the directive

    `#include <filename> -`
    - Searches standard library for file
    - Use for standard library files

    `#include "filename"`
    - Searches current directory, then standard library
    - Use for user-defined files
- Used for
  - Loading header files (**`#include <iostream>`**)
  - Programs with multiple source files to be compiled together
  - Header file - has common declarations and definitions (classes, structures, function prototypes)
    - **`#include`** statement in each file

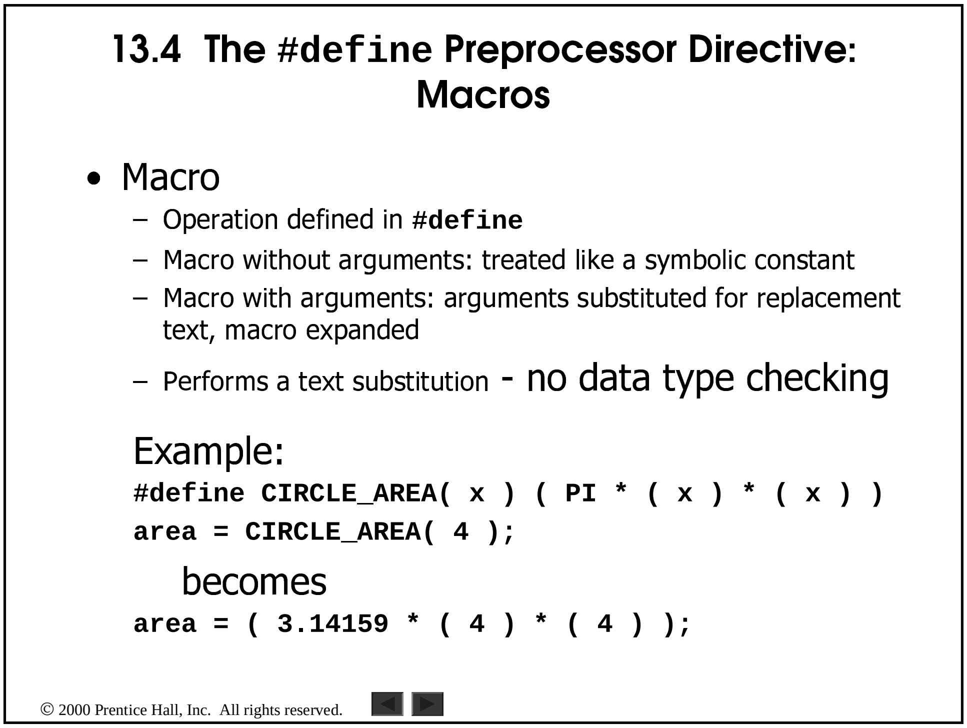


13.3 The `#define` Preprocessor Directive: Symbolic Constants

- **`#define`**
  - Preprocessor directive used to create symbolic constants and macros.
- Symbolic constants
  - When program compiled, all occurrences of symbolic constant replaced with replacement text
- Format

  `#define identifier replacement-text`
  - Example: **`#define PI 3.14159`**
  - everything to right of identifier replaces text

    `#define PI = 3.14159`
    - replaces "**`PI`**" with " **`= 3.14159`**", probably results in an error
  - Cannot redefine symbolic constants with more **`#define`** statements



13.4 The #define Preprocessor Directive: Macros

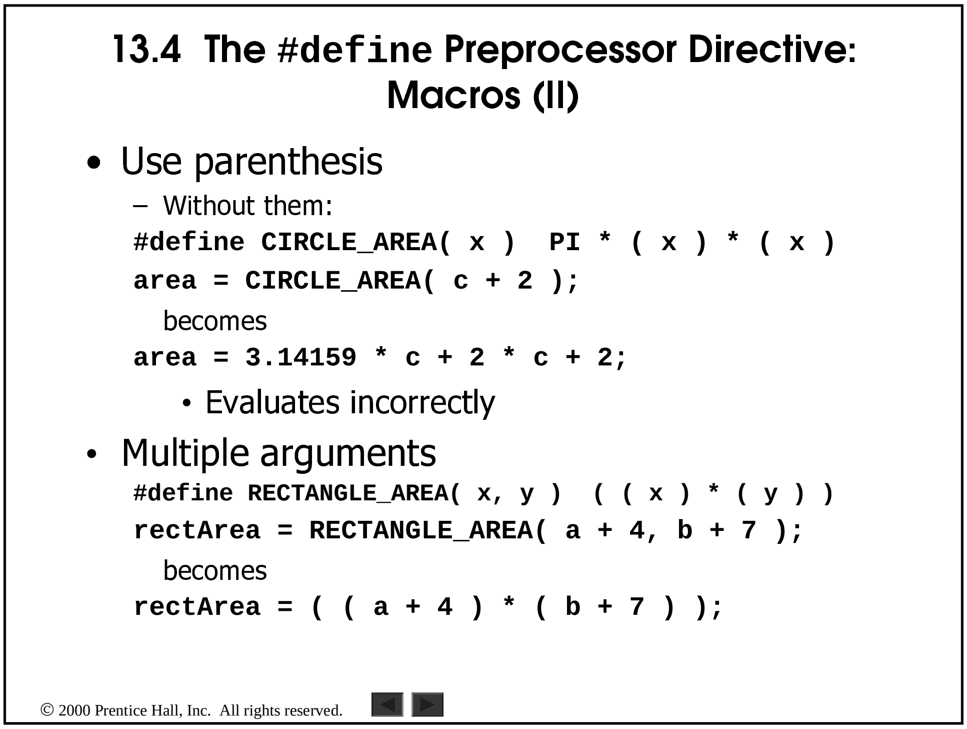
- Macro
  - Operation defined in `#define`
  - Macro without arguments: treated like a symbolic constant
  - Macro with arguments: arguments substituted for replacement text, macro expanded
  - Performs a text substitution - no data type checking

Example:

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

becomes

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



13.4 The #define Preprocessor Directive: Macros (II)

- Use parenthesis
  - Without them:

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

becomes

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

  - Evaluates incorrectly
- Multiple arguments

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

becomes

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

13.4 The #define Preprocessor Directive: Macros (III)

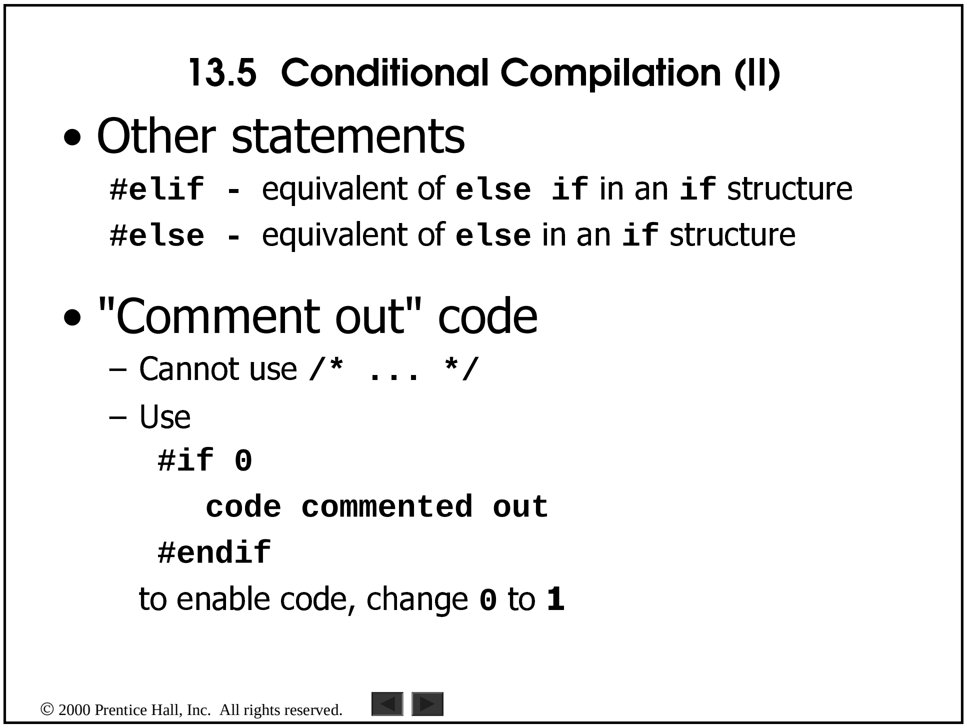
- **#undef**
  - Undefines a symbolic constant or macro, which can later be redefined

13.5 Conditional Compilation

- Conditional compilation
  - Control preprocessor directives and compilation
  - Cast expressions, **sizeof**, enumeration constants cannot be evaluated
- Structure similar to **if**

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

  - Determines if symbolic constant **NULL** defined
    - If **NULL** is defined, **defined(NULL)** evaluates to **1**
    - If **NULL** not defined, defines **NULL** as **0**
  - Every **#if** ends with **#endif**
  - **#ifdef** short for **#if defined(*name*)**
  - **#ifndef** short for **#if !defined(*name*)**



13.5 Conditional Compilation (II)

- Other statements

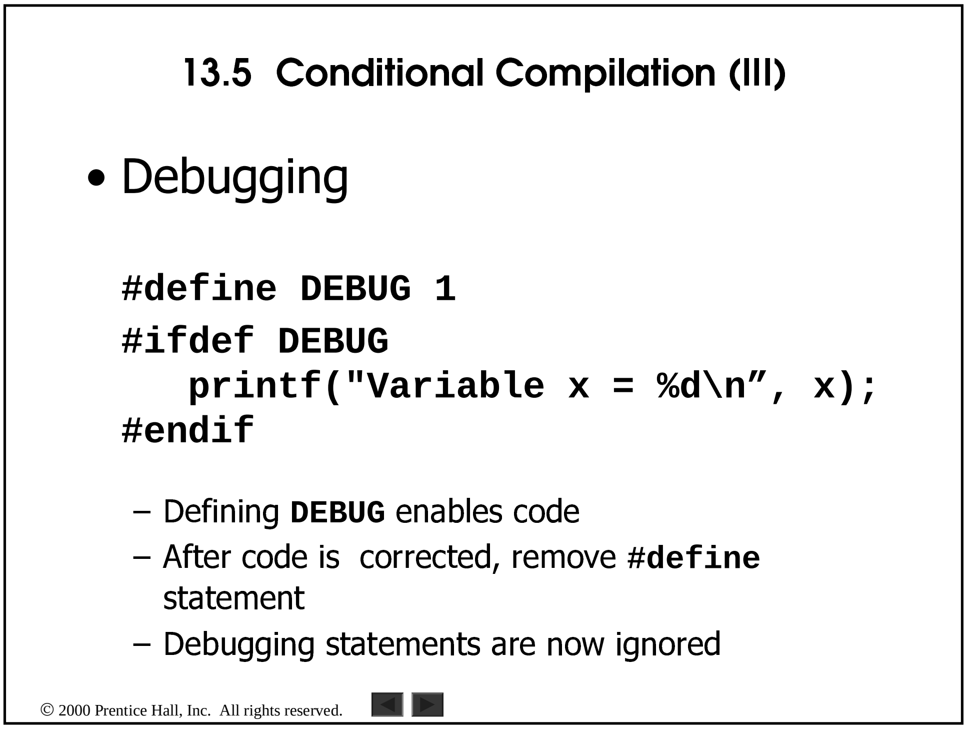
  #elif - equivalent of **else if** in an **if** structure

  #else - equivalent of **else** in an **if** structure

- "Comment out" code
  - Cannot use `/* ... */`
  - Use

```
#if 0
   code commented out
#endif
```

to enable code, change **0** to **1**



13.5 Conditional Compilation (III)

- Debugging

```
#define DEBUG 1
#ifdef DEBUG
   printf("Variable x = %d\n”, x);
#endif
```

- Defining **DEBUG** enables code
- After code is corrected, remove **#define** statement
- Debugging statements are now ignored

13.6 The #error and #pragma Preprocessor Directives

- **#error** *tokens*
  - Tokens - sequences of characters separated by spaces
    - **"I like C"** has 3 tokens
  - Prints message and tokens (depends on implementation)
  - For example: when **#error** encountered, tokens displayed and preprocessing stops (program does not compile)
- **#pragma** *tokens*
  - Implementation defined action (consult compiler documentation)
  - Pragmas not recognized by compiler are ignored

13.7 The # and ## Operators

- **#** - “stringization”
  - Replacement text token converted to string with quotes

```
#define HELLO( x ) printf(#x “\n”)
```

Notice #

```
HELLO(John);
```

becomes

printf(“John”, “\n”);

- Strings separated by whitespace are concatenated in C, so we have:

printf(“John”, “\n”);

- **## - concatenation**
  - Concatenates two tokens

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

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

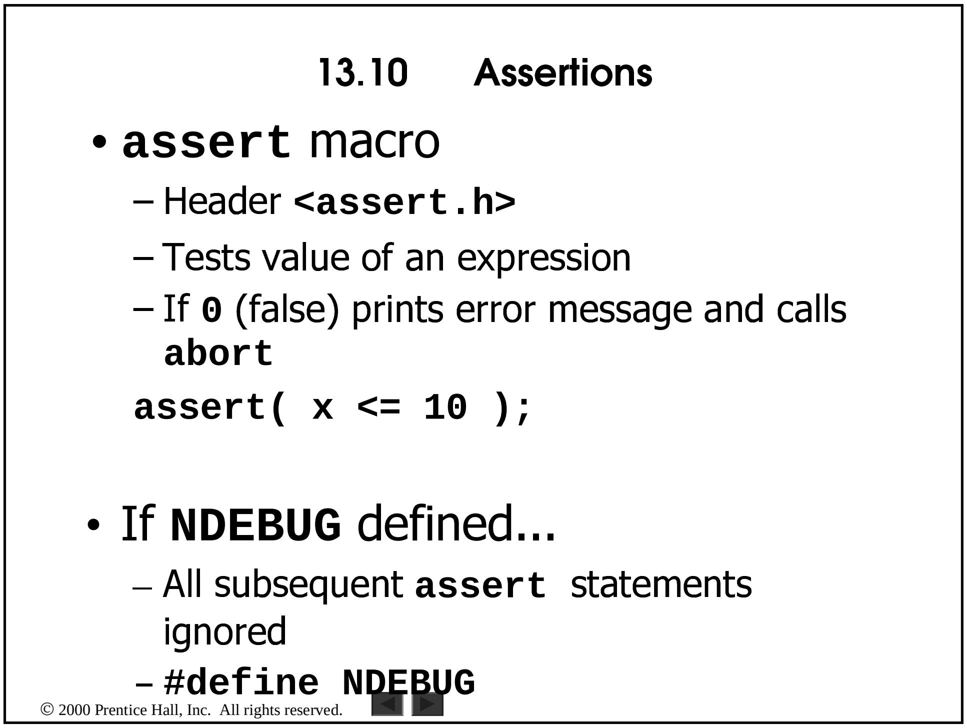
13.8 Line Numbers

- **`#line`**
  - renumbers subsequent code lines, starting with integer value
  - file name can be included
- **`#line 100 "myFile.c"`**
  - Lines are numbered from **`100`** beginning with next source code file
  - For purposes of errors, file name is **`"myfile.C"`**
  - Makes errors more meaningful
  - Line numbers do not appear in source file

13.9 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"`**).



13.10 Assertions

- **`assert`** macro
  - Header **`<assert.h>`**
  - Tests value of an expression
  - If **`0`** (false) prints error message and calls **`abort`**

```
assert( x <= 10 );
```

- If **`NDEBUG`** defined...
  - All subsequent **`assert`** statements ignored
  - **`#define NDEBUG`**