UMBC
Training Centers
6996 Columbia Gateway Drive
Suite 100
Columbia, MD 21046
Tel: 443-692-6600
http://www.umbctraining.com

INTERMEDIATE C PROGRAMMING

Course # TCPGR3001
Rev. 10/27/2016

This Page Intentionally Left Blank

Course Objectives

- At the conclusion of this course, students will be able to:
  - Master the use of pointers in a wide variety of problems.
  - Use sophisticated pointer techniques to solve problems involving advanced data structures such as lists, stacks, queues and trees.
  - Choose from a wide variety of data structures to implement the most efficient solution to a problem.
  - Apply the proper optimization technique to your C code.
  - Apply many portability techniques to your C code.
  - Use bit manipulation techniques for efficient solutions to problems.
  - Write programs which emphasize modern program design techniques which emphasize code reuse.
  - Write powerful C programs which make "calls" directly into the UNIX operating system through the use of system calls.
  - Decrease development time in writing applications through a more thorough understanding of sophisticated concepts in C.

This Page Intentionally Left Blank

Table of Contents

Chapter 1:

Review of C and Aggregate Data Types

Data Types

- int

  10 256 -3245
  2456456L 32457U
  'a' '?' '\007'

- float

  23.45F 345.31f

- double

  25.67 45.324E+05

- derived data types

```
enum weekend { sat, sun};


struct point {   union hash {
int x;   long int index;
int y;   char name[4];
};   };
```

- typedefs

```
typedef enum weekend WKEND;
typedef struct point POINT;
typedef union hash HASH;
```

Data Types

- For integers

`int`	normally 2 or 4 bytes
`long`	normally 4 or 8 bytes
`short`	normally 2 or 4 bytes
`unsigned`	sizeof (int) bit manipulation
`unsigned`	long sizeof(long)
`unsigned`	short sizeof(short)

- For small integers or characters

`char`	sizeof(char)
`unsigned char`	sizeof(char)
`signed char`	for small ints

- For decimal numbers

`float`	numbers with fractional parts or very large or small
`double`	twice the precision of float
`long double`	more precision/magnitude

- For aggregate data

`struct`	create your own types
`union`	alternative interpretation
`enum`	symbolic constants

- Synonyms for all data types

`typedef`	create new name

Operators

- Logical

```
if (! ( c >= 'a' && c <= 'z'))
```

code;

- Relational

```
if( a == b || c != d)
```

code;

- Assignment

```
x *= a + b;
```

- Bitwise

```
unsigned a,b,c;
a = ~b | c;
```

- Structure Member

```
struct point a, *p;
p = &a;
printf("%d,%d\n", p -> x, a.y);
```

- Conditional

```
x = a > b ? a: b;
```

Operators

- Arithmetic

```
+    -    *    /    %
```

- Logical

```
!         &&   ||
```

- Relational

```
==   !=   >    >=   <    <=
```

- Assignment

```
=    +=   -=   *=   \=   etc
```

- Bitwise

```
~    &    |    ^    >>   <<
```

- Structure Member Pointer

```
.         ->        &    *
```

- Miscellaneous

```
++   --   ?:   ,    sizeof   (type)   [ ]  ( )
```

Control Flow Constructs

- if

```
if(condition) {
    code for true
}
```

- if...else

```
if(condition) {
    code for true
}
else {
    code for false
}
```

- if...else if...else

```
if(condition_1) {
    code for condition_1 true
}
else if(condition_2) {
    code for condition_2 true
}
    :
    :
else if(condition_n) {
    code for condition_n true
}
else {
    code for all conditions false
}
```

Control Flow Constructs

- for

```
for(initial_expr ; test_expr ; modify_expr) {
    code for the body of the loop
}
```

- while

```
while(test_expr) {
    code for the body of the loop
}
```

- do while

```
do {
    code for the body of the loop
} while(test_expr);
```

Control Flow Constructs

- switch

```
int c;

c = getchar( );

switch(c) {
case 'a':
case 'A':
case '1':
    add( );
    break;

case 'd':
case 'D':
case '2':
    delete( );
    break;

default:
    printf("Choose 'add' or 'delete'\n");
}
```

Control Flow Constructs

- switch - general syntax

```
switch(integer_test_expression) {
case integer_constant_expr_1:
case integer_constant_expr_2:
        .....
        .....
        body of statements to be executed when
        test_expression = any case expression.
        break;
case integer_constant_expr_1a:
case integer_constant_expr_2a:
        .....
        .....
        body of statements to be executed when
        test_expression = any case expression.
        break;
        .....
        .....


default:
     code to be executed when no
     cases match test_expression
}
```

- Note that break is optional and, if omitted, program will slide through to code for next case.
- Conditional expression for a switch must evaluate to an `int` or be a parameter of type `int`, `short` or `char`.

Aggregate Data Types

- Arrays

```
int numbers[MAX_NUMBERS];
char one_line[STRING_SIZE];
```

- Array of arrays

  Each `strings`[i] is an array of `STRING_SIZE` many characters.

```
char strings[NUM_STRINGS][STRING_SIZE];
```

- Array of pointers

```
char *pointers[NUM_STRINGS];
```

- Pointer initialization

```
char line[STRING_SIZE];
char *p = line, *q, *r;

q = p;
r = malloc(STRING_SIZE);

fun(p);

fun(char *v)
{

}
```

Arrays

- Arrays - collections of single elements traversed with subscripts or pointers
- Array of arrays
- Pointer arrays
- Uses of pointers
  - traversing arrays
  - altering arguments through function calls
  - efficiency
  - creating high level data structures such as lists, queues, stacks, trees
- Initialization of pointers
  - with definition
  - explicitly through assignment
  - relationship between arguments and parameters
  - returned value from `malloc`

Arrays

- Observe the following analogy.

```
int x;                          // x is an integer variable
x = 5;                          // 5 is an integer constant

char *pc;                       // pc is a pointer variable
char line[MAX_CHARS];           // line is a pointer constant
pc = line;                      // legal
```

- You cannot change a constant.

```
5 = x;                          // is illegal
line = pc;                      // is illegal
```

- You can change a variable.

```
pc++;                           // is legal
x++;                            // is legal
```

- In order to change a variable through a function call, the function must be sent the address of the variable.

```
fun(line);      // line is an address by definition
fun(&x);        // you must explicitly send address of x
```

Arrays

- An array is:
  - the name of a contiguous portion of memory, say x
  - the address in memory of the beginning of x
  - an address constant
- A pointer variable is a variable, which can hold the address of another variable.
- The connection between an array and a pointer is that both of them represent an address. However, an array is a **pointer constant** while a **pointer variable** can be modified.

Structures

- A structure

```
struct point {
int x;
int y;
};
```

- A structure within a structure

```
struct window {
struct point upper_left;
struct point lower_right;
};
```

- Referencing structures

```
main( )
{
struct point apoint;
struct window awindow;

apoint.x = 5;
apoint.y = 10;

awindow.upper_left.x = 5;
awindow.upper_left.y = 10;

awindow.upper_left = apoint;
}
```

Structures

- Structures
  - aggregate data type
  - usually has many elements called members
  - programmer created type
- Structures should model aggregate data from problem space.
  - a payroll record
  - a window
  - a job in an operating system
- Using structures in a program
  - provide a description of the structure (header file)
  - allocate storage for one - by definition or by **`malloc`**
  - fill in the defined storage
- Used heavily in dynamic data structures such as lists, trees, stacks, queues

Structures

- Structures may be assigned to one another if they have the same data type.

```
struct date a,b, *ps;
struct payroll person, manager;


a = b;                      // ok
person = manager            // ok
a = manager;                // illegal
a = manager.birthday;       // ok

ps = &a;
b = *ps;                    // ok
```

typedef

- In programs using structures, the keyword `struct` must be repeated often.
- There is a facility called `typedef`, which allows you to create a new name for an existing type. A `typedef` can:
  - Simplify programs
  - Aid in documenting programs
  - Give the appearance of new types
- Conventionally, a `typedef` is named using upper case.
- The lines below give new names for existing types.

```
/* LENGTH and WIDTH are additional
      names for int
*/
typedef int LENGTH, WIDTH;

/* STRING is a new name for char **/
typedef char * STRING;

/* ARRAY is a new name for array
   of 10 integers
*/
typedef int ARRAY[10];
```

- A `typedef` is usually placed in a header file.

typedef

- With these `typedef`'s in place, you could code:

```
int main()
{
        /*  a, b, x, and y are of the same type */

        int a, b;
        WIDTH x;
        LENGTH y;

        /*  p, pc are of the same type  */

        char *p;
        STRING pc;

        /*  x, r are of the same type  */

        int x[10];
        ARRAY r;

        ...
}
```

- A `typedef` is often used as new name for a structure type. For example:

```
typedef struct employee WORKER, *PWKR;

WORKER company[10];       // Array of structures
PWKR   answer;            // Pointer to structure
```

Structures - Example

- Assume the following is contained in **payroll.h**.

```
#define NAMESIZE 30

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

struct payroll {
     char name[NAMESIZE];
     float pay;
     struct date birthday;
     struct date promotion;
     char dept[10];
};

typedef struct payroll RECORD;
```

Structures - Example

- Assume the following is contained in your driver program.

```
#include "payroll.h"
#include <stdlib.h>
main( )
{
RECORD person;          // a struct payroll variable
RECORD bank[100]        // array of 100 struct payroll
RECORD *pts;            // a pointer to a struct payroll
RECORD fill_1( );       // function returning RECORD
RECORD *fill_2( );      // function returning RECORD *

pts =  (RECORD *) malloc(sizeof(RECORD));
if( pts == NULL)
error( );
person = fill_1( );
pts = fill_2( );
}
```

Structures - Example

- Structure members are referenced with the syntax:
  - structure_name.member_name

```
struct date feb14;
struct payroll person;

feb14.year = 1992;
feb14.month = 2;
feb14.day = 14;

person.pay = 42500.00;
strcpy(person.name,"tom smith");
person.birthday = feb14;
```

- Pointers to structures need to use pointer qualification:
  - pointer -> structure_member

```
struct date *pts;
struct payroll *pp;

pts = &feb14;
pts -> year = 1992
pts -> month = 2;
pts -> day = 14;

pp = &person;
pp -> pay = 42500.00;
pp -> birthday = feb14;
```

Unions

- A driver program that instantiates a union

```
#include "hash.h"
#include <stdio.h>
main( )
{
int probe;
char table[NUM_STRINGS][MAX_CHARS];
union hash string;
fgets(string.name, MAX_CHARS, stdin);
probe = string.index % NUM_STRINGS;
if ( strcmp(table[probe],string.name) == 0 )
    printf("found %s\n", string.name);
else
....
}
```

- The header file **`hash.h`**

```
#define MAX_CHARS 4
#define NUM_STRINGS 100
union hash {
char name[MAX_CHARS];
long int index;
};
```

Unions

- Unions
  - use structure syntax
  - have different interpretations of the same storage
  - `sizeof(union) >= sizeof(largest interpretation)`
  - have specialized uses such as:
    - variant records
    - non homogeneous arrays
  - create possible portability problems
- Using **unions** in a program
  - provide a description of the union - (header file)
  - allocate storage for one
  - refer to storage by any of the collection of types in the union

Bitfields

- Assume the bitfield structure is defined in **`info.h`**.

```
struct information {
unsigned int is_male: 1;
unsigned int is_married: 1;
unsigned int how_many_kids: 5;
unsigned int is_manager: 1;
};
```

- A driver program which instantiates and uses a bitfield structure.

```
#include "info.h"
main( )
{
struct information person;

/* code to fill up the person would go here */

if ( person.is_male && person.is_married )
function( );
person.how_many_kids++;
    ....
}
```

Bitfields

- Bitfields
  - allow easy access to portions of the wordsize
  - provide an alternative to bit manipulation operators
  - use structure syntax for easier access than bit manipulation operators
  - may create portability problems
- Using bitfields in a program
  - provide a description of the bit field type (usually kept in a header file)
  - allocate storage for one
  - refer to storage by structure syntax

Enumerations

- Assume enumeration is defined in **`sports.h`**.

```
enum sports
{ baseball, football, basketball, hockey };
/*  baseball = 0;  */
enum products { shoes = 10, rackets, shirts };
/*  rackets = 11;  */
```

- A driver program which instantiates and uses an enumeration

```
#include "sports.h"
main( )
{
enum sports activity;
enum sports next_sport( );

activity = next_sport(baseball);
if ( activity == basketball )
function( );
....

}
```

Enumerations

- Enumerations
  - are modeled after pascal although not as strongly implemented (no **succ** or **pred** functions)
  - makes programs easier to read
  - slightly stronger than #define mechanism
  - are implemented as small integers
  - are symbolic constants whose values start with 0 unless otherwise specified and are incremented by 1.
- Using the **enum** type in a program
  - provide a description of the type together with possible values (usually kept in a header file)
  - allocate storage for one

Exercises

1. Write a small program, which prints the sizes of various integers on your machine.
2. For the following definitions and code, give the value of each expression below:

```
char *pc;
char lines[10][20];

strcpy(lines[0], "mike");
pc = lines[0];
```

a) `sizeof(lines)`
b) `sizeof(lines[1]))`
c) `sizeof(lines[1][3])`
d) `strlen(lines[0])`
e) `sizeof(pc)`

3. Write the declarations for a structure which contains a union (of a character array and a long integer), a structure of bit fields (containing a day, month and year), and an ordinary structure (containing a first name, last name, and initial). (Hint: First, write the declarations for each of the contained items.)
4. How do you access:

   a) the char array portion of the union inside the structure?
   b) the day portion of the bit field in the structure?
   c) the middle initial portion of the initial portion of the structure?

Chapter 2:

Building Larger Programs

Compiling Over Several Files

- The source for one executable can be spread out over several files.
  - How is a variable, which is defined in one file, known to another file?
  - How is a variable, which is defined in one file, kept private to that file?
  - Over what part of the program can a variable be accessed?
- There are advantages in keeping a function in a file by itself.
  - The function can be compiled once.
  - It can be linked whenever needed.
  - It can be placed in a library.
- The `scope` of a variable refers to that portion of a program where the variable may be accessed.
  - Block scope
  - Function scope
  - File scope
  - Program scope

Function Scope

- Variables have `function` scope if they are either:
  - Non-`static` variables defined inside of a function
  - Parameters.
- Storage for these variables is reserved when the function in which they are defined is executed.
  - Storage is deallocated when the function terminates.
  - The keyword `auto` can be used for local variables that are not parameters.
  - Local variables can only be referenced inside the function in which they are defined.
- Parameters begin life with the value of the argument that they represent.
  - Other local variables have unknown beginning values.
- `a, b,` and `c` have function scope. They are referred to as **local** variables.

```
double hypot(double a, double b)
{
    double c;

    c = sqrt(a * a + b * b);
    return(c);
}
```

File Scope

- A variable has file scope if it is defined in a file before a function.
  - This is a less likely place to define a variable.
  - It is the source of many bugs.
  - Yet, there are some good uses for file scoped variables.
- Functions, which do not call one another but need to share data, should use file-scoped variables.

```
int v;          /* file scope */
int x[100];     /* file scope */
int s;            /* file scope */

void push(int value)
{
      // body of push
}

int pop()
{
         int s;   // file scope 's' not visible here
}

void clear()
{
      // body of clear
}
```

- `v` and `x` can be referenced from any function in this file. s is known to the entire file except in `pop`. Inside `pop`, another variable (also named `s`) has local scope.

Program Scope

- Program scope means that a variable is known to all files of a program.
  - Program scope differs from file scope when there is more than one file being linked.
- This is usually not desirable because a change to a variable in one file can have an effect on the same variable in another file.
- Often, the programmer wants privacy for a file scoped variable.
- The keyword `static` makes a file scoped variable private to that file.

Program Scope

- `x` has program scope.

main.c

```
extern int x;

main()
{
   fun1();
   fun2();
}

fun1()
{
   fun3();
   fun5();
}
```

one.c

```
extern int x;

fun2()
{
   fun1();
   fun4();
}

fun3()
{
   ...
   ...
}
```

two.c

```
Int x;

fun4()
{
    fun5();
    fun3();
}

 fun5()
{
    ...
    ...
}
```

- If the three files above are compiled and linked together, there will be `one` storage location named `x`.
  - Among all these files, exactly one of them must have a definition for `x`. The others need an extern declaration for `x`.
- Any of the variables above can be made private to a particular file by using the `static` keyword.

```
static int x;

main()
{
   fun1();
   fun2();
}

fun1()
{
   fun3();
   fun5();
}
```

```
static int x;

fun2()
{
   fun1();
   fun4();
}

fun3()
{
   ...
   ...
}
```

```
static int x;

fun4()
{
   fun5();
   fun3();
}

fun5()
{
   ...
   ...
}
```

Local `static`

- Whether they are `static` or not:
  - file-scoped variables have initial values of 0; and
  - only one initialization for a file scoped variable is permitted.
- The `static` keyword has another important use.
  - Local static variables retain values across function invocations. They are variables that are initialized once, regardless of how many times the function is invoked.
- A local `static` variable is initialized when the program is loaded.

Local `static`

starter.c

```
 1. #include <stdio.h>
 2.
 3. #define HOWMANY 5
 4.
 5. void starter();
 6.
 7. main()
 8. {
 9.     int i;
10.
11.     for ( i = 0; i < HOWMANY; i++)
12.            starter();
13. }
14.
15. void starter()
16. {
17.     /*
18.            Static storage is initialized with 0
19.            by default.  Here we do it explicitly
20.     */
21.      static int first = 0;
22.
23.      if(first == 0)
24.      {
25.            first = 1;
26.            printf ("Only once\n");
27.      }
28.      else
29.      {
30.            printf ("All but the first time\n");
31.      }
32.
33.      printf ("Every time\n");
34. }
```

`register` and `extern`

- There are four keywords associated with storage classes.
  - `auto` Local variables
  - `static` Either for privacy or to remember values across function invocations
  - `extern` Link to variable defined elsewhere
  - `register` Attempt to speed up the program
- An `extern` declaration is used to link a variable to a definition in another file.
- The `register` keyword is used to try to increase the speed of a function.
  - A machine register will be used if available.
  - Index variables are good candidates for `register` variables.
  - Only `int, char,` or pointer types can be `register` variables.

register and extern

- A `register` variable is an attempt to speed up a function.

```
void fun()
{
        register int i;

        for (i = 0; i < 1000; i++)
            x[i] = y[i] + z[i];
}
```

- `extern` is used in order to access a file-scoped variable defined in another file.

file1.c

```
main()
{


}

fun1()
{
   extern int r;
}
```

file2.c

```
int r;

void fun2()
{

}

int fun3()
{

}
```

- A more common use for `extern` is to access a system variable.

Object Files

- If each function is placed in a separate file, the functions can be compiled separately and then linked into any application that needs them.
- Every compiler has an option that directs the compiler to build an object file rather than to build an executable file. The resultant object file is named with the suffix `.o` on UNIX machines and `.obj` on Windows machines.
- On UNIX for example, one might proceed as follows:

```
$ gcc #c fun1.c
$ gcc #c fun2.c
$ gcc prog.c fun1.o fun2.o
```

Example

prog.c

```
 1. #include <stdio.h>
 2.
 3. void fun1();
 4. void fun2();
 5.
 6. int x;
 7.
 8. main()
 9. {
10.      printf("MAIN: x = %d\n", x);
11.      fun1();
12.      printf("MAIN: x = %d\n", x);
13.      fun2();
14.      printf("MAIN: x = %d\n", x);
15. }
```

fun1.c

```
1. #include <stdio.h>
2.
3. int x;
4.
5. void fun1()
6. {
7.      x++;
8.      printf("FUN1: x = %d\n", x);
9. }
```

fun2.c

```
1. #include <stdio.h>
2.
3. int x;
4.
5. void fun2()
6. {
7.      x++;
8.      printf("FUN2: x = %d\n", x);
9. }
```

Libraries

- The previous strategy is cumbersome with a large number of files. A better strategy is to use libraries.
- Each operating system has a utility (the librarian).
  - Compile each function as before.
  - Place the object files in the library.
- Place library name on the command line when building an executable.
- When your C compiler was installed, many support tools were also installed including:
  - The librarian utility
  - The actual libraries
- To build a library, one might proceed as follows.
  - For example, on UNIX:

```
$ gcc #c fun1.c
$ gcc #c fun2.c
$ ar r mylib.a fun1.o fun2.o
```

- Building executables:

```
$ gcc main.c mylib.a
$ gcc othermain.c mylib.a
```

The C Loader

- When you execute the C compiler, several steps occur.
  - Preprocessor
  - Compiler
  - Loader
- Loader looks for functions referenced but not supplied.
- In order to resolve external references, the loader looks various places in the following order.
  - In the source file being compiled
  - In any files being linked
  - In the Standard C Library
  - In libraries named on the command line
- For any function not resolved by the above search, an error is printed.
- A program with a misspelled function name would produce the following error.

```
int main()
{
      printf("Hello\n");
      print("Loader\n");      /* oops */
}
```

- Loader will see `print` as a referenced function whose code was not supplied and will report an error.

```
unresolved external: _print
```

Header Files

- Header files support applications and libraries. Header files contain information needed by many programs.
- Header files save you typing. You can use the `#include` feature of the preprocessor to import header files. Header files should not contain executable statements.
- There is a relationship between header files and libraries.
- When you use a function from a library, you should include the header file, which supports that library.
  - The header file will contain the prototype(s) for the function(s) from that library that you are using.
  - This reduces the likelihood of errors, which could result if you provided your own prototypes.

Header Files

- Information typically found in header files
  - `#include` statements
  - `#define` statements
  - Other preprocessor directives
  - Function prototypes
  - Structure descriptions
  - `typedef`'s
- Each library on your system will have at least one header file that supports it. Examples:

LIBRARY	HEADER FILE

Standard C	`stdio.h`
	`stdlib.h`
	`string.h`

Math	`math.h`

Exercises

1. Take any program that you have written and spread it out so there is one function per file. Compile and execute the program.
2. Simulate an array by providing two functions, `get` and `put`. Place these functions in the same file and define an array externally to these two functions. Compile this file separately.

```
int array[1000];
void put(int value, int position)
{

}

int get(int position)
{

}
```

   - Programs wishing to use an array can do so by making calls to `get` and `put` in the following way rather than by using subscripts.

```
main()
{
     int x;
     put(10,5);   //  array[5] = 10;

     x = get(5);  //   x = array[5];
}
```

3. Test your program by reading some numbers typed at the keyboard and printing them in reverse order.
   - Note also that you could put in array bounds checking in `get` and `put`.

This Page Intentionally Left Blank

Chapter 3:

Functions

Function Fundamentals

- Function Example

```
/*  function prototype   */
double average(int *table, int num_elements);

#define ARRAY_SIZE 100
int main()
{
     double avg;
     int numbers[ARRAY_SIZE];
     /*
      fill up array numbers
     */
     /*  function invocation  */
     avg = average(numbers, ARRAY_SIZE);
}

/* function definition */
double average(int *array, int limit)
{
     double total = 0.0;
     int i;
     for( i = 0; i < limit; i++)
          total += array[i];
     return( total / limit );
}
```

- Note that the parentheses for the return statement are optional when a value is being returned. The following are equivalent!

```
return( total / limit );
return total / limit;
```

Function Fundamentals

- Functions are the basic building blocks of C programs.
  - Generality
  - Modularity
  - Reuse
  - Readability
- Three pieces of information must associate correctly.
  - Function prototype
  - Function invocation
  - Function definition

Function Prototypes

- Function prototypes are usually kept in a header file.
  - Prototypes for I/O functions in the standard library are kept in `stdio.h`.
  - Prototypes for the math functions in the standard library are kept in `math.h`.
  - Prototypes for the string functions in the standard library are kept in `string.h`.
- Prototype example

```
void initialize(int *table, int num_elements);
```

Function Invocation and Definition

- Invocations must originate from within any function.
- The definition for a function can be:
  - in the same file from which it is invoked;
  - in a different file from where it is invoked; or
  - compiled into a library and linked with the invoking function.
- The definition for a function consists of two parts.
  - the function header
  - the function body
- The function body consists of all of the declarations for variables followed by all of the executable statements.
- Example of function invocation and definition

```
#define ARRAY_SIZE 100
int main()
{
        int numbers[ARRAY_SIZE];
        /*  function invocation  */
        initialize(numbers, ARRAY_SIZE);
}

/* function definition */
void initialize(int *array, int limit)
{
        int i;
        for( i = 0; i < limit; i++)
            array[i] = 0;
}
```

Function Prototypes

- A prototype tells the compiler the types of the arguments and the return type of the function.

```
double average(int *table, int num_of_elements);
```

- Prototype arguments do not need names, just types, thus the following two are equivalent.

```
double average(int *table, int num_of_elements);
double average(int *, int);
```

- A function declaration merely tells the compiler the return type of the function (K & R Compiler Style).

```
double average();
```

- In the absence of either a prototype or a declaration, the following is a difficult bug to find!

demo.c

```
 1. #define STRING_SIZE 100
 2.
 3. int main()
 4. {
 5.         char string[STRING_SIZE];
 6.         double x;
 7.
 8.         gets(string);
 9.         x = atof(string);
10.         printf("%f\n", x);
11. }
```

Function Prototypes

- Function prototypes convey two pieces of information.
  - the returned type of the function
  - the types of the parameters
- Note that the names of the parameters are not required in the prototypes. If they are provided, it is purely for documentation (readability) purposes.
- Before ANSI C, there were no prototypes. There were only function declarations. A function declaration simply tells the compiler the return type of the function.
- Function prototypes are not required by ANSI C, but you are encouraged to use them! They allow the compiler to do type checking. They provide documentation for the interface to functions.
- In the absence of a prototype or a declaration, the compiler assumes the function returns int.

Subprogram Examples

- Examples of functions from other languages

```
X = SQRT(Y);
J = SQRT( SQUARE(X) + SQUARE(Y));
Z = SQUARE(X) * CUBE(Y) * 100;
```

- Examples of subroutines from other languages

```
CALL SORT(ARRAY, AMOUNT);

CALL DISPLAY(VALUES);
```

- A C function returning a value

```
avg = average(numbers, howmany);
```

- C functions not returning a value

```
/*  sorting an array  */
sort(numbers, amount);

/*  printing a structure  */
print(person);
```

Functions Returning a Value

- Some programming languages offer two subprogram types.
  - a function
    - arguments and parameters are related
    - must have a return statement
    - one expression (value) is returned
    - the function name represents the value returned
    - can be used as an expression in a computation
  - a subroutine
    - arguments and parameters are related
    - subroutine name does not represent a value
    - there is no return statement
- C only has one subprogram type - the function.
- Generally, if a function is to compute a value, it will be given a return type.
- If the function is to behave like a subroutine from other languages, it will be given the return type of void.

Return Value Considerations

- We wish to write a function to compute the sum, the average, and the variance.

```
/*       INVOCATION   */
stats(numbers, howmany, &sum, &mean, &variance);

/*       PROTOTYPE    */
void stats(int*, int, int*, double*, double*);
```

- Premature return from a `void` function

```
void fun(int a)
{
     if( a < 0 ) {
          report();
          return;
     }
     if( a == 0 ) {
          display();
          return;
     }
     /* rest of function */
}
```

Return Value Considerations

- If a function is to "return" more than one item, it can be done in a number of ways. The simplest way is to provide an out argument for each value that you want the function to compute.
- `void` functions need not supply a return statement.
- If a function returning `void` can return from more than one place, then a return statement provides a simple solution.
  - Note: In this situation, it is the flow of control, which is returned and not a value.

```
return;
```

- For functions which actually return a value, the return is coded as:

```
/* parenthesis not necessary */
return(expression);
```

- Expression is converted to the return type of the function.

Recursive Functions

- Iterative solution for computing factorial of an integer

```
int fact(int p)
{
    int res = 1;
    if( p <= 1 )
        return(1);
    for ( ; p > 1; p--)
        res *= p;
    return(res);
}
```

- Recursive solution for computing factorial of an integer

```
int fact(int p)
{
    if( p <= 1 )
        return(1);
    return(p * fact(p - 1));
}
```

Recursive Functions

- A recursive function is one that invokes itself.
- Compare the two functions, each of which computes a factorial.
- The iterative solution
  - takes more local space
  - is faster
- The recursive solution
  - takes more stack space
  - is easier to read
  - runs more slowly

Recursive Functions

- Fibonacci Sequence

```
int fib(int p)
{
     if ( p <= 1 )
          return(p);
     return( fib(p - 1) + fib(p - 2));
}
```

- Notice that fib(0) and fib(1) require one iteration but:

FIB	INVOCATIONS
2	3
3	5
4	9
5	15

Recursive Functions

- To see why recursion is usually slower than iterative versions of the same function, notice the recursive version of the Fibonacci sequence.

0,1,1,2,3,5,8,13,21
0,1,2,3,4,5,6, 7, 8

fib(n) = n	for n < 2
fib(n) = fib(n - 1) + fib(n - 2)	for n >= 2

- A more efficient solution to this problem is shown in the efficiency chapter.

Recursive Functions

- Quicksort - strategy
  - For the following data elements:

```
10  20  25  55  35  42  11  5  29  2  30
```

  - Choose (10) as the pivotal element.
  - Start a pointer from the left and find an element larger than (10). 20 is the element.
  - Start a pointer from the right and find an element smaller than (10). 2 is the element.
  - Swap these two elements.

```
10  (2)  25  55  35  42  11  5  29  (20)  30
```

  - Repeat this process until they cross.

```
10  (2)  (5)  55  35  42  11  (25)  29  (20)  30
```

  - Swap the pivot element.

```
(5) (2)  (10)  55  35  42  11  (25)  29  (20)  30

-------------  -----  ------------------------------------------------------------
LOWER    PIVOT                    HIGHER
```

  - The data is now partitioned about the pivotal element.
  - Call the quicksort function recursively on each of the two partitions.

Recursive Functions

- Quicksort is the most studied sorting algorithm. For many kinds of data, quicksort is the fastest sort.
  - Many embellishments
  - Difficult to code
- Divide and conquer strategy
  - Select a partitioning element (say the first element). Call this P.
  - Partition the data so that
    - all elements to the left of P are lower than P
    - all elements to the right are greater than P
- Recursively apply the same algorithm to each of the two partitions!

Recursive Functions

- Code for quicksort

quicksort.c

```
 1. #define SIZE 20
 2. void quick(int *, int, int);
 3. int main()
 4. {
 5.     int a[SIZE];
 6.     fill(a,SIZE);
 7.     print(a,SIZE);
 8.     quick(a, 0, SIZE - 1);
 9.     print(a,SIZE);
10. }
11. void quick(int *p, int low, int high)
12. {
13.     if ( low < high) {
14.          int lo = low;
15.          int hi = high + 1;
16.          int elem = p[low];
17.          for ( ; ; ) {
18.             while(p[++lo] < elem && lo < hi)
19.                  ;
20.             while(p[--hi]>elem  &&  high>=lo)
21.                  ;
22.             if ( lo < hi)
23.                  swap(&p[lo], &p[hi]);
24.             else
25.                  break;
26.          }
27.          swap(&p[low], &p[hi]);
28.          quick(p, low, hi - 1);
29.          quick(p, hi + 1, high);
30.    }
31. }
```

Evaluation of Function Arguments

- Example Program 1

prog1.c

```
 1. int add(int a, int b)
 2. {
 3.     return a + b;
 4. }
 5.
 6. int main()
 7. {
 8.     int i = 5, j = 5, x, y;
 9.
10.     x = add(i, i++)
11.     y = add(j++,j);
12.     printf("%d %d\n", x, y);
13. }
```

- Example Program 2

prog2.c

```
1. int main()
2. {
3.     int i = 5, j = 5;
4.
5.     printf("%d %d\n", i++, i);
6.     printf("%d %d\n", j, j++);
7. }
```

Evaluation of Function Arguments

- The order in which C compilers evaluate function arguments is undefined. If you are not aware of this, difficult bugs can creep into your program. It is sometimes difficult to see the effects of this.
- Avoid the following.

```
power(i, i++);

printf("%d %d\n", i, i++)

x = f(i) + g(i++);
```

Variable Number of Arguments

- `printf` has a variable number of arguments.

```
printf("hello");
printf("%s is a string\n", line);
printf("%s has length %d\n",line,strlen(line));
```

- A portable way of handling varying number of arguments

varargs.c

```
 1. #include <stdarg.h>
 2.
 3. /*     Prototype for sum:  Note the ... and no
 4.     type info after first arg  */
 5.
 6. int sum(int, ...);
 7.
 8. main()
 9. {
10.      printf("%d\n", sum(3, 7, -2, 4));
11.      printf("%d\n", sum(5,1,2,3,4,5));
12. }
13.
14. int sum(int n, ...) /* args have no names */
15. {
16.     va_list arg;     /* arg refers to each
17.                              arg in turn */
18.      int i, s;
19.
20.      va_start(arg, n);     /* arg points to first arg */
21.
22.      for (i = s = 0; i < n ; i++ ) {
23.         s += va_arg(arg, int); /*return next arg*/
24.      }
25.
26.      va_end(arg);                  /* cleanup */
27.      return(s);
28. }
```

Variable Number of Arguments

- Most functions are invoked with a consistent number of arguments although the need may arise to write a function whose argument count may vary across invocations.
- Different systems handle argument passing in different ways, and therefore the previous solution does not work! In the function definition, code more parameters than the number of the actual arguments.
- However, ANSI has provided a standard interface. The details are kept in the file `stdarg.h`. The details in the header file pertain to the specific system on which you are working.

Initialization

- Local static initialization example

static.c

```
 1. #define MAX 10
 2. #define LOOPS 100
 3.
 4. void fun(void);
 5.
 6. int main()
 7. {
 8.      int i;
 9.
10.      for(i = 0; i < LOOPS; i++) {
11.            fun();
12.     }
13. }
14.
15. void fun()
16. {
17.      static int x = 0;
18.
19.      if((x % MAX) == (MAX - 1)) {
20.            printf("%d\n", x);
21.      }
22.      x++;
23. }
24.
```

Initialization

- Local variables are auto by default. They have unknown initial values.
- External variables are extern by default. They have 0 as their initial value.
- All static variables are initially 0 by default.
- Local static variables remember values between function invocations.
- In pre ANSI C, aggregates (arrays and structures) cannot be initialized, unless they are external or static. This is sensible from an efficiency point of view. The ANSI standard has relaxed this condition.

Exercises

1. Write a recursive version of the function add, which returns the sum of two positive integer arguments.

2. Write a function `record()` which takes two integer parameters. The function tracks jockey's wins (first place), places (second place), and shows (third place). Use `#defines` (or `enums`) liberally. Be careful of your choice of values for the constants. Use a static `int` array local to the `record()` function.

   The call `record(1, WIN)` records a WIN for player 1.
   The call `record(2, SHOW)` records a SHOW for player 2.

   ```
   record(ALL, PRINT);   // prints all records
   record(ALL, CLEAR);   // clears all records
   record(ALL, POINTS);  // show points for all
                         // records
   ```

   WIN = 4 points, PLACE = 2 points, SHOW = 1 point

   There is a starter file for this exercise in the `starters` directory.

3. Suppose a function `power` is invoked as shown below. What is the value of x? Try it!

   ```
   int i = 2, y, x;

   y = power(2,3);     // y = 8
   y = power(2,5);     // y = 32
   x = power(i, i++); // x = ???
   ```

4. Give a good example of the use of an external static variable.

This Page Intentionally Left Blank

Chapter 4:

Bit Manipulation

Characteristics of Bitwise Problems

- We wish to track many binary characteristics.
  - An Employee Profile
    - male or female
    - supervisor or not
    - veteran or not
    - college educated or not
    - married or single
    - minor or not
    - etc...

Defining the Problem Space

- There is a class of problems whose solutions are best accomplished by manipulating the bits of a variable.
- Usually typified by tracking many on/off values for these problems, a solution is provided by the data type `unsigned int`. A variable of this type occupies the same amount of storage as an `int`.

```
sizeof(unsigned int) == sizeof(int)
```

- `unsigned int` has no sign bit. You can use the `unsigned` type to store numbers although they must be positive or zero. Each piece (**bit**) of information could be stored in a single `int`, but this would be wasteful.
- A more economical approach is to pack the information into an `unsigned int` a bit at a time. The solution becomes a matter of accessing single bits.

Bitwise Operators

- The following truth table summarizes the rules for the bitwise logical operators.

OP	OP	|	&	^
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

- Other bit operators

  a and b in memory (Assume 16 bit wordsize.)

```
unsigned int a,b,c;

a = 055;          // OCTAL CONSTANT
b = 0x44;         // HEX CONSTANT



a    =    0 000 000 000 101 101
b    =    0 000 000 001 000 100

a | b    0 000 000 001 101 101
a & b    0 000 000 000 000 100
a ^ b    0 000 000 001 101 001

~a       1 111 111 111 010 010
a >> 3   0 000 000 000 000 101
b << 3   0 000 001 000 100 000
```

Bitwise Operators

- To access and manipulate single bits, C provides six **bitwise operators.**
  - bitwise or |
  - bitwise and **&**
  - bitwise exclusive or **^**
  - right shift **>>**
  - left shift **<<**
  - one's complement **~**
- The **bitwise logical or (|)**, obeys the following rule.
  - Bits are compared one at a time.
  - Each resultant bit is a **1** bit if either or both operand bits have the value **1**.
- The **bitwise logical and (&)**, obeys the following rule.
  - Bits are compared one at a time.
  - Each resultant bit is a **1** bit if both operand bits have the value **1.**
- The **bitwise logical exclusive or (^)**, obeys the following rule.
  - Bits are compared one at a time.
  - Each resultant bit is a **1** bit if either **but not both** operand bits have the value **1**.

Readability Aids

- Define the following constants for readability.

```
#define IS_MALE  01
#define IS_MAN   02
#define IS_VET   04
#define HAS_DEG  010
#define IS_MAR   020
#define IS_MIN   040
```

- To assign the characteristic of being **married**:

```
unsigned int profile = 0;
profile = IS_MAR;
```

- To add the characteristic of being male:

```
profile = profile|IS_MALE;
```

- To assign the characteristics of a degreed, married, male vet:

```
profile = (HAS_DEG|IS_MAR|IS_MALE|IS_VET);
HAS_DEG =  0 0     0 0 1  0 0 0    =     010
IS_MAR  =  0 0     0 1 0  0 0 0    =     020
IS_MALE =  0 0     0 0 0  0 0 1    =     001
IS_VET  =  0 0     0 0 0  1 0 0    =     004
-----------------------------------------
profile =  0 0     0 1 1  1 0 1
```

Assigning Bit Values

- We often need to assign a value to a specific bit position.
  - Using octal or hexadecimal values makes this easier
  - Octal constants must have a leading 0 (zero)
  - Hex constants must have a leading 0x or 0X
- The **bitwise logical or (|)** is a simple way of adding in a bit value without disrupting other bits.
- Examples

```
#define IS_MALE  01       #define IS_MAN    02
#define IS_VET   04       #define HAS_DEG  010
#define IS_MAR   020      #define IS_MIN   040
main()
{
     unsigned int profile = 0;
     profile |= (HAS_DEG|IS_MAR|IS_MALE|IS_VET );
     if( profile&HAS_DEG )
          printf("HAS DEGREE\n");

     if( profile&IS_VET )
          printf("IS VET\n");

     if( profile&(IS_MAN|HAS_DEG))
          printf("IS MAN OR HAS DEGREE\n");

     if((profile&(IS_MAN|HAS_DEG))==(IS_MAN|AS_DEG))
          printf("IS MAN AND HAS DEGREE\n");

     if((profile&(IS_MAR|HAS_DEG))==(IS_MAR|AS_DEG))
          printf("IS MAR AND HAS DEGREE\n");
  }
```

Assigning Bit Values

- A bitwise function
  - The **wordlength** function computes the number of bits for an int regardless of the machine

```
int wordlength(void)
{
     unsigned int x = ~0; // x GETS ALL one BITS
     int count = 0;
     while( x != 0 ) {    // ANY one BITS REMAINING?
          count++;        // YES
          x >>= 1;        // SHIFT 1 TO THE RIGHT

     }
     return(count);       // NO
}
```

Writing Bitwise Functions

- Other kinds of problems use bit manipulation features.
- The size of an `int` varies from machine to machine. Therefore, it may be convenient to write a function that computes the number of bits in a word.
- The **>>** operator shifts bits to the right.
  - Right shifting an `int` is machine dependent.
- The **<<** operator shifts bits to the left.
  - Bit positions fill with **zeros** when you shift an `unsigned`.

Circular Shifts

- Example

cshift.c

```
 1. unsigned int rcshift(unsigned int, int);
 2. unsigned int lcshift(unsigned int, int);
 3. int wordlength(void);
 4. main()
 5. {
 6.   unsigned int n = 0xffff;
 7.   int num = 4;
 8.   printf("%d\n", wordlength());
 9.     // %x is for hexadecimal
10.   printf("BEFORE: %x\n",n);
11.   n = rcshift(n,num);
12.   printf("AFTER RIGHT:  %x\n",n);
13.   n = lcshift(n,num * num);
14.   printf("AFTER LEFT:   %x\n",n);
15. }
16. unsigned int rcshift
17. (unsigned int number, int howmany)
18. {
19.   int i;
20.   unsigned mask = ~0;
21.   mask = ~(mask >> 1);
22.   for( i = 0; i < howmany; i++) {
23.         if ( number & 01)
24.             number = (number >> 1) | mask;
25.         else
26.             number >>= 1;
27.   }
28.   return(number);
29. }
30. unsigned int lcshift
31. (unsigned int number, int howmany)
32. {
33.   unsigned int r;
34.   r = rcshift(number, wordlength() # howmany);
35.   return(r);
36. }
```

Circular Shifts

- A **circular shift** fills the original with the shifted off bit.

```
ABCDEFGH // original
BCDEFGH0 // original LEFT SHIFT 1 BIT
BCDEFGHA // original LEFT CIRCULAR SHIFT 1
```

- C has no circular shift operators.
- You must implement them with your own functions.
  - We show `rcshift(n,num)` which right circular shifts the variable **n, num** bits.
  - We also show `lcshift(n,num)` which left circular shifts **n, num** bits.
  - Notice the use of the **wordlength** function in the **lcshift** function.

Character Information Array

- Provide symbolic names for each character type.

```
#define A        001     // ALPHABETIC
#define D        002     // DIGIT
#define L        004     // LOWER CASE
#define U        010     // UPPER CASE
#define H        020     // HEX DIGIT
#define O        040     // OCTAL DIGIT
#define P        0100    // PUNCTUATION
#define SPACE    0200    // SPACE
#define CT       0400    // CONTROL CHAR
```

```
unsigned int chars[] = {
/*  0 */
CT,CT,CT,CT,CT,CT,CT,CT,CT,SPACE,SPACE,CT,CT,CT,CT,CT
/* 16 */
CT,CT,CT,CT,CT,CT,CT,CT,CT,CT,CT,CT,CT,CT,CT,CT,
/* 32 */  CT,CT,CT,CT,CT,CT,CT,CT,SPACE,CT,CT,CT,P,CT,CT,CT,
/* 48 */
D|H|O, D|H|O, D|H|O, D|H|O, D|H|O, D|H|O, D|H|O, D|H|O, D|H,
D|H, P, P, P, P, P, P, P,
/* 65 */
U|A|H, U|A|H, U|A|H, U|A|H, U|A|H, U|A|H,
/* 71 */
U|A, U|A, U|A, U|A, U|A, U|A, U|A, U|A, U|A, U|A, U|A, U|A,
U|A, U|A, U|A, U|A, U|A, U|A, U|A, U|A, P, P, P, P, P, P,
/* 97 */
L|A|H, L|A|H, L|A|H, L|A|H, L|A|H, L|A|H,
/*103 */
L|A, L|A, L|A, L|A, L|A, L|A, L|A, L|A, L|A, L|A, L|A, L|A,
L|A, L|A, L|A, L|A, L|A, L|A, L|A, L|A
};
```

- Finally, define **macros** in terms of the array `chars`.

```
#define IS_ALPHA(C) chars[C] & A
#define IS_DIGIT(C) chars[C] & D
#define IS_SPACE(C) chars[C] & SPACE
```

Direct Lookup

- The Standard C Library consists of many functions that do character testing.
  - Some implement these as functions.
  - Others are implemented as macros.
- One look up technique can be provided by the following scheme:
  - Each character has a unique 7 bit configuration that can be considered a number.
  - For example, the character 'a' has the configuration.
  - **1 000 001** = 65 DECIMAL
- Provide an `unsigned int` array large enough to hold information about every character.
  - At position **65**, place symbolic information about the character **'a'**.

```
U | A | H          // UPPER, ALPHA, HEX
```

  - Do the same for each character in the set.

Mapping With Bits

- Checkerboard example
  - The occupied array is mapped as follows.

R/C	7	6	5	4	3	2	1	0
0	7	6	5	4	3	2	1	0
1	15	14	13	12	11	10	9	8
2	23	22	21	20	19	18	17	16
3	31	30	29	28	27	26	25	24
4	39	38	37	36	35	34	33	32
5	47	46	45	44	43	42	41	40
6	55	54	53	52	51	50	49	48
7	63	62	61	60	59	58	57	56

- Many bit problems require more than the number of bits in an `unsigned int`.
  - the status of the squares of a checkerboard
  - use an **array of** `unsigned char`
- We also provide a **print_binary** routine to **see** the actual board.

Mapping With Bits

- Example of mapping with bits

checkers.c

```
 1. #define HOW_MANY_RANDOMS   10
 2. #define N 64
 3. #define BPW 8                    //BITS PER WORD
 4. #define ELEMENTS (N/BPW)         //ARRAY ELEMENTS
 5. void binary_print(unsigned char);
 6. main( )
 7. {
 8.   unsigned char occupied[ELEMENTS];
 9.   int i, number, row, bit;
10.   for ( i = 0; i < ELEMENTS; i++)
11.        occupied[i] = 0;
12.   // generate a few random numbers
13.   for (i = 0; i < HOW_MANY_RANDOMS; i++) {
14.        number = rand() % N;
15.        row = number / BPW;    //ROW NUMBER
16.        bit = number % BPW;     //BIT NUMBER
17.        occupied[row] |= 1 << bit;
18.   }
19.   for ( i = 0; i < ELEMENTS; i++)
20.        binary_print(occupied[i]);
21. }
22. //print an unsigned char as binary digits
23. void binary_print(unsigned char element)
24. {
25.   int i;
26.   char output[BPW + 1];
27.   unsigned mask = 1;
28.   for (i = 0; i < BPW; i++){
29.        if(element & mask)
30.             output[BPW - 1 # i] = '1';
31.        else
32.             output[BPW - 1 # i] = '0';
33.        mask <<= 1;
34.   }
35.   output[i] = '\0';
36.   printf("%s\n", output);
37. }
```

Radix Sort

- Consider the following data and binary equivalences:

13	12	10	4	11	7	9	2
1101	1100	1010	0100	1101	0111	1001	0010

- Separate (s) according to right most bit and collect (c).

	(S)	(C)	(S)	(C)	(S)	(C)	(S)	(C)
B	110**0**	1100	11**0**0	1100	1**0**01	1001	**0**010	= 2
I	101**0**	1010	01**0**0	0100	1**0**10	1010	**0**100	= 4
N	010**0**	0100	11**0**1	1101	0**0**10	0010	**0**111	= 7
0	001**0**	0010	10**0**1	1001	1**0**11	1011	====	
	====	1101	====	1010	====	1100	**1**001	= 9
B	110**1**	1011	10**1**0	0010	1**1**00	0100	**1**010	= 10
I	101**1**	0111	00**1**0	1011	0**1**00	1101	**1**011	= 11
N	011**1**	1001	10**1**1	0111	1**1**01	0111	**1**100	= 12
1	100**1**		01**1**1		0**1**11		**1**101	= 13

- Note that the final collection is in sorted order.

Radix Sort

- Radix sorting differs widely from most sorts.
- It is more difficult to code.
  - Look at the right most bit, bit 0, of each key
  - Place all those whose value is zero in bin zero
  - Place all those whose value is one in bin one
  - Collect the two bins with one on the bottom
  - Repeat the process looking at bit number 1
  - Repeat the process looking at bit number 2 etc

Radix Sort

radixsort.c

```
 1. #define SIZE 15
 2. int wordlength(void);
 3. void radix(int *v, int size);
 4. void print(int * array, int size);
 5. main()
 6. {
 7. int data[]={46,30,82,90,56,17,95,15,48,26,4,58,71,79,92};
 8. printf("ORIGINAL:\n");
 9. print(data, SIZE);
10. radix(data, SIZE);
11. printf("SORTED:\n");
12. print(data, SIZE);
13. }
14. void radix(int *v, int size)
15. {
16.   int temp[SIZE], t_index, v_index, i, k = 0;
17.   unsigned mask;
18.   int wordsize = wordlength() - 1;
19.   // LOOP(WORDSIZE -1) TIMES
20.   while (k < wordsize)
21.   {
22.        t_index = v_index = 0;
23.        // mask for bit manipulation
24.        mask = 1 << k;
25.        // separate into v and temp
26.        for ( i = 0; i < size; i++){
27.             if ((v[i] & mask) == 0)
28.                  v[v_index++] = v[i];
29.             else
30.                  temp[t_index++] = v[i];
31.        }
32.        //combine
33.        for(i = 0; i < t_index; i ++)
34.             v[v_index++] = temp[i];
35.        // just for aid in displaying the output
36.        //print(v, SIZE);
37.        k++;
38.   }
39. }
```

Radix Sort

- The **three** lines below show the data before and after the program. The middle line shows the data after one pass through the **radix** sort.
- Notice that all the even numbers are followed by all the odd numbers.

ORIGINAL: 46 30 82 90 56 17 95 15 48 26 4 58 71 79 92

ONE PASS: 46 30 82 90 56 48 26 4 58 92 17 95 15 71 79

SORTED: 4 15 17 26 30 46 48 56 58 71 79 82 90 92 95

Exercises

1. Write a program which queries the user for the action **on** or **off**. Then ask for a **bit** number **b**. The program should perform the specified action on bit number **b.**

```
$ prog
action-> on
bit number 2
0000000000000010
action-> on
bit number 4
0000000000001010
action-> off
bit number 2
0000000000001000
action-> quit
$
```

2. Write a function which determines if an unsigned integer is a bit palindrome.

10101	YES
11010	NO
11011	YES

There is a starter file for this exercise in the `starters` directory.

This Page Intentionally Left Blank

Chapter 5:

Pointers (Part 1)

Common Pointer Constructs

- Pointer post-incrementation

```
*p++
```

  - Fetch what is pointed to.
  - Increment the pointer.

- Pointer pre-incrementation

```
*++p
```

  - Increment the pointer.
  - Fetch what it points to.

- Incrementing or decrementing a pointer is only sensible when pointing inside an array.

Common Pointer Constructs

- Pointers are used to:
  - modify arguments sent to functions;
  - traverse arrays;
  - create sophisticated data structures; and
  - make programs efficient.
- Fundamental pointer axiom

  *For any **type**,*

  if

  ```
  type *p, var;
  p = &var;
  ```

  then

  ```
  *p == var
  ```
- The pointer construct **p++**
  - sensible when **p** points at an array element
  - advances **p** to point at next array element
  - has a counterpart **p--**
  - is an example of pointer arithmetic
- The pointer construct ***p++**:
  - represents the value that **p** points to
  - increments **p** after ***p** is used

Pointer Arithmetic

- Examples of pointer arithmetic:

```
#define MAX 10

int *end, *begin, numbers[MAX], sum = 0;

begin = numbers;
end   = numbers + MAX;    // pointer arithmetic

while (begin < end) {     // pointer comparison
    if (*begin < 0) {
        break;
    }
    sum += *begin++;      // pointer arithmetic
}

if (begin == numbers + MAX)  // ptr arithmetic
    printf("no negatives\n");
}
```

- To see relationship between **end** and **numbers + MAX**

```
int *end, numbers[MAX];

// assume numbers is at 2000 in memory
// assume sizeof(int) == 4

(1) end ==  numbers  +  MAX
(2) end ==  2000     +  10 * sizeof(int)
(3) end ==  2000     +  10 * 4
(4) end ==  2000     +  40
(5) end ==  2040
```

Pointer Arithmetic

- Pointer arithmetic has several forms.
  - pointer + integer => pointer result
  - pointer - integer => pointer result
  - pointer - pointer => integer result
- C supports specific pointer types rather than generic pointers.
  - automatic scaling for pointer arithmetic
  - used exclusively for array processing
- Example using pointers

pointer.c

```
 1. #include <stdio.h>
 2. #include <malloc.h>
 3.
 4. int main()
 5. {
 6.     int *iPtr;
 7.     void *ptr = malloc(10);
 8.
 9.     iPtr = ptr;
10.     *iPtr = 10;
11.
12.     printf("%d %p\n", *iPtr, iPtr);
13.     printf("%d %p\n", *ptr, ptr);
14.     printf("Sizeof *iPtr is %d\n", sizeof(*iPtr));
15.     printf("Sizeof *ptr is %d\n", sizeof(*ptr));
16. }
```

Binary Search

- Another example of using pointers

binary.c

```
 1. #define SIZE 5
 2.
 3. int * binary(int * table, int size, int value);
 4.
 5. int main()
 6. {
 7.     // array must be sorted for a binary search
 8.
 9.     int data[] = {10, 12, 34, 45, 65};
10.    int lookup = 21;
11.     int * found = binary(data, SIZE, lookup);
12.
13.     if(found) {
14.          printf("%d was found\n", *found); {
15.     } else {
16.          printf("%d was not found\n", lookup);
17.    }
18. }
19.
20. int * binary(int * table, int size, int value)
21. {
22.     int *begin = table;
23.    int *end = begin + size -1, *mid;
24.
25.     while (begin <= end) {
26.          mid = begin + (end - begin) / 2;
27.          if (*mid > value) {
28.             end = mid - 1;
29.          } else if(*mid < value) {
30.             begin = mid + 1;
31.          } else {
32.             return mid;
33.       }
34.     }
35.     return NULL;
36. }
```

Binary Search

- There are various methods for searching tables.
  - Linear Search
  - Binary Search
  - Hashing
- If a data structure has too many elements, then linear search is too costly.
- You can lower the time it takes to find a value by using a technique known as binary search.
- With binary search, the data must be kept in sorted order.
  - By probing at the middle element, you can determine which half to search.
  - Use the same strategy on the remaining half.

Changing a Pointer through a Function Call

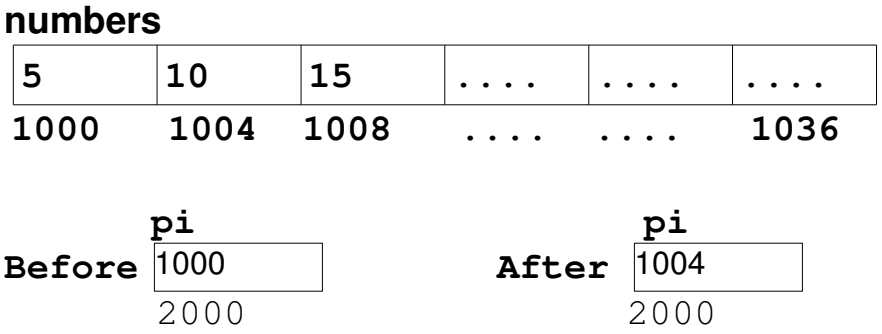
- Recall the first important use of pointers to swap two integers.
  - Must send the addresses of the integers

```
void swap(int *first, int *second)

int main( )
{
     int a, b;

     // MUST PASS ADDRESSES
     swap(&a, &b);

}
```

- How do you change a pointer argument?
  - Send the address of the pointer.
  - Parameter must be pointer to pointer.

Changing a Pointer through a Function Call

- In the `main` function:




```
void change(int **p);

int main()
{
     int numbers[10] = { 5, 10, 15 };
     int *pi;

     // SET pi TO POINT TO numbers
     pi = numbers;

     change(&pi);   // pass address of pointer
}

void change(int **p)
{
     *p = *p + 1;      // change pi in main
}
```

Processing Arrays With Pointers

- Traditional way of stepping through an array

```
int numbers[MAX], i, sum = 0;

for (i = 0; i < MAX; i++) {
     sum += numbers[i];
}
```

- Using a pointer can be more efficient.

```
int numbers[MAX], *p, sum = 0;

for (p = numbers; p < numbers + MAX; p++) {
   sum += *p;
}
```

```
int numbers[MAX];
int *p = numbers, *end = numbers + MAX;

while(p < end) {
     printf("%d ", *p++);
}
```

- A two dimensional array

```
int values[ROWS][COLS],

*p = &values[0][0];
*end = &values[ROWS - 1][COLS - 1];

while(p <= end) {
     printf("%d ", *p++);
}
```

Simulating Higher Dimensional Arrays

- Arrays and pointers are both addresses.
  - Arrays can use pointer notation.
  - Pointers can be subscripted.
  - Each of the above requires the same address calculation.
- Simulating a two dimensional integer array with `malloc`
  - Define a pointer to an integer pointer.

```
int **p;
```

  - If the array is to be x rows, then allocate x many integer pointers.

```
p = malloc(x * sizeof(int *));
```

  - Allocate the space for each row (say y elements per row).

```
p[i] = malloc(y * sizeof(int));
```

- `p` is a pointer to the first of `x` integer pointers.
  - `*p` is of type `int *`
  - `p[0]` is also of type `int *`
  - `p[0][j]` is the `j`th integer in the "row" pointed to by `p[0]`

Simulating Higher Dimensional Arrays

- Arrays can use pointer notation.

```
int number[MAX], i, sum = 0;
for( i = 0; i < MAX; i++)
     *(numbers + i) = 0;
```

- Pointers can be subscripted.

```
int numbers[MAX], *pi = numbers;
for( i = 0; i < MAX; i++)
     pi[i] = 0;
for( i = 0; i < MAX; i++)
     numbers[i] = 0;
```

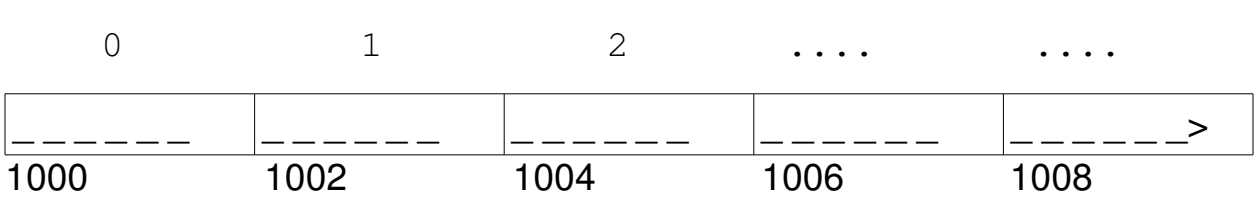
twodim.c

```
 1. #include <stdlib.h>
 2. #include <stdio.h>
 3.
 4. #define ROWS 3
 5. #define COLS 5
 6.
 7. int main()
 8. {
 9.     int **p, i, j;
10.
11.     p = malloc(ROWS * sizeof(int *));
12.     if ( p == NULL) {
13.          printf("Memory allocation error\n");
14.       exit(1);
15.    }
16.     for( i = 0; i < ROWS; i++) {
17.          p[i] = malloc( COLS * sizeof(int));
18.          if (p[i] == NULL) {
19.             printf("Memory allocation error\n");
20.           exit(2);
21.       }
22.     }
23.     for ( i = 0; i < ROWS; i++)
24.          for ( j = 0; j < COLS ; j++)
25.        p[i][j] = 10 * i + j;
26. }
```

Two Dimensional Arrays

- The following is a fundamental notion for arrays.

```
vector[i] == *(vector + i)

vector[3] = 10;
*(vector + 3) = 10
```



- Apply the notion twice for two dimensional arrays.

```
int numbers[4][3];

numbers[2][1] = 20;
```

Then `numbers[2][1]` is resolved in the following way:

```
*(numbers + 2)[1]              // ONCE
*(*(numbers + 2) + 1)          // TWICE
```

- Assuming numbers starts at address 5000, then:

Expression	Value	Type
`numbers`	5000	ptr to array of 3 `int's`
`numbers + 2`	5012	ptr to array of 3 `int's`
`*(numbers + 2)`	5012	ptr to `int`
`*(numbers + 2) + 1`	5014	ptr to `int`
`*(*(numbers + 2) + 1)`	20	`int`

Two Dimensional Arrays

- Declaring a two dimensional array:

```
#define ROWS 5
#define COLS 10
#define NO_STRINGS 10
#define LENGTH 50

int numbers[ROWS][COLS];
char lines[NO_STRINGS][LENGTH];
```

- Arrays of arrays
  - `numbers` and `lines` are actually single dimensioned arrays whose members are also arrays.
- The following are correct:

```
void fun(int *p);

int vector[COLS];
int *p, (*pv)[COLS];

fun(vector);
fun(numbers[i]);

p = vector;
pv = numbers;
```

- The following are incorrect:

```
fun(numbers);
p = numbers;
```

Complex Declarations

- Simple

```
int pi;            // int

int *pi;           // pointer to an int

int vals[MAX]     // array of ints

int f();          // function returning int
```

- Combined

```
int (*pt)[MAX];  // one pointer to MAX ints

int *ptrs[MAX];  // array of pointers to int

int *fun();      // function returning pointer to int

int (*fun)();    // pointer to function returning int
```

- Harder

```
int (*f[M])();   // M pointers to functions returning
                 // int
int *(*f)();     // pointer to function returning
                 // pointer to int

int *(*f[M])();  // array of M pointers to functions
                // returning pointer to int
```

Complex Declarations

- Use the hierarchy of C operators.

  `()` Comes before
  `[]` Comes before
  `*`

```
int (*pt)[MAX];    // one pointer to MAX ints

int *ptrs[MAX];    // array of pointers to int
```

- This often results in reading a declaration in the order left-right-left-right etc.

```
int *(*f[M])();      // array of M pointers to
                     // functions returning
                     // pointer to int
```

`f[M]`	Array of M
`*f[M]`	pointers to
`(*f[M])()`	function returning
`int *(*f[M])()`	pointer to int

Sorting with Large Records

- All the sorting methods shown earlier exchanged data when out of order.
  - Expensive for large records
  - Impossible for varying length records
- A better technique is to **compare** records **but exchange pointers**.
- The actual sorting method is unimportant. We use **selection** sort.

surrogate.h

```
 1. #define NAMESIZE 20
 2.
 3. struct records {
 4.        char name[NAMESIZE];
 5.        int age;
 6. };
 7.
 8. typedef struct records R, *PR;
 9.
10. void print(PR *, int);
11. void sort(PR *, int);
12. int input_rec(PR);
13. int get_the_data(PR *, int);
```

Sorting with Large Records

surrogate.c

```
 1. #include <stdio.h>
 2.
 3. #include "surrogate.h"
 4.
 5. void sort(PR *pts, int size) {
 6.     PR temp;
 7.     int i, j, min;
 8.
 9.     for ( i = 0; i < size; i++) {
10.          min = i;
11.          for( j = i + 1; j < size; j++) {
12.             if(pts[j] #> age <  pts[min] #> age) {
13.                min = j;
14.                temp = pts[min];
15.             pts[min] = pts[i];
16.             pts[i] = temp;
17.          }
18.       }
19.     }
20. }
21.
22. int get_the_data(PR *pts, int amount) {
23.     R temp;
24.     int i = 0;
25.
26.     while(input_rec(&temp) !=  EOF) {
27.          if( i > amount)
28.             return(#1);
29.          pts[i] = (PR) malloc(sizeof(R));
30.          if( pts[i] == NULL)
31.             return(#1);
32.          *(pts[i]) = temp;
33.          i++;
34.     }
35.     return(i);
36. }
```

Sorting with Large Records

surrogate.c (continued)

```
37. int input_rec(PR temp) {
38.     char line[100];
39.
40.     printf("Enter name and age: ");
41.     if (gets(line) == NULL)
42.          return(EOF);
43.
44.     sscanf(line,"%s %d", temp #> name, &temp #> age);
45.     return(! EOF);
46. }
47.
48. void print(PR *p, int howmany) {
49.     int i;
50.
51.    for(i= 0; i < howmany; i++)
52.          printf(" %s %d\n", p[i] #> name, p[i] #> age);
53. }
```

surrogatemain.c

```
 1. #include "surrogate.h"
 2.
 3. #define NUM 100
 4.
 5. int main( )  {
 6.    PR record_pointers[NUM];
 7.     int number;
 8.
 9.    number = get_the_data(record_pointers, NUM);
10.     if(number > 0) {
11.          sort(record_pointers, number);
12.          print(record_pointers, number);
13.     }
14. }
```

Exercises

1. Write a program which performs a frequency count of the words in a file. Use the following "parallel" arrays to solve the problem.

```
char words[NUMBER_OF_WORDS][WORD_LENGTH + 1];
int counts[NUMBER_OF_WORDS];
```

- Note: A simple way to get words from an input file is to use the function `scanf` as shown below:

```
#include <stdio.h>

#define MAXLINE 100

int main()
{
     char line[MAXLINE];

     //  scanf returns EOF at end of file

     while(scanf("%s", line) != EOF {
          printf("%s\n", line);          // Print word
    }
}
```

2. Write a function called `sort3` that takes three integer points as arguments and sorts (in ascending order) the three integers pointed to. Here is some code to get you started:

```
void sort3 (int *ptr1, int *ptr2, int *ptr3);

int main()
{
    int a = 7, b = 2, c = 4;

    sort3(&a, &b, &c);
    printf("After: a = %d, b = %d, c = %d\n", a, b, c);

    // a = 2, b = 4, c = 7
}
```

This Page Intentionally Left Blank

Chapter 6:

Pointers (Part 2)

Dynamic Memory Allocation

- Arrays cause storage to be allocated before execution (static storage allocation).
  - The size of an array is a worst-case scenario.
  - The size of an array cannot be altered.
- Storage can be allocated during program execution (dynamic storage allocation).
  - It usually results from program conditions or user requests.
- Dynamic storage is obtained by invoking the standard C library function named `malloc`.
- `malloc` manages a huge pool of storage and hands out pieces of memory to satisfy program requests. A pointer to the requested space is returned unless the request cannot be satisfied, in which case the value `NULL`, defined in `stdio.h`, is returned.
- Therefore, `malloc` returns a pointer. The prototype for this function is in `malloc.h`, so you must include the header file or code the following:

```
void *malloc(int);
```

- The type `void *` is a generic pointer which can be assigned to any other pointer.

Dynamic Memory Allocation

malloc-demo.c

```
 1. #include <stdio.h>
 2. #include <stdlib.h>
 3. #include <malloc.h>
 4.
 5. #define MAX 100
 6.
 7. int main()
 8. {
 9.      int len, n;
10.      char input[MAX], *pc;
11.
12.      printf("How many input chars? ");
13.      fgets(input, MAX, stdin);   /*  get input  */
14.
15.      n = atoi(input); /*  convert to number  */
16.      pc = malloc(n + 1);   /*  ask for space  */
17.
18.      if (pc == NULL) {     /*  is there any?  */
19.            printf("malloc: no room\n");   /*  No  */
20.            exit(1);
21.      }
22.
23.      printf("Input up to %d chars: ", n);  /*  Yes  */
24.      fgets(pc, n + 1, stdin);   /*  input the string  */
25.     len = strlen(pc);
26.
27.      printf("You input %d chars\n", len);
28.
29.      free(pc);             /*  free the space  */
30.
31.      return 0;
32. }
33.
```

Dynamic Memory Allocation

- `malloc` returns a pointer to the first of the requested bytes.
- Therefore, you can use this space in the same way that you can use an array. In fact, `malloc` gives you a dynamic array.

```
char *pc;
int len;

pc = malloc(100);
fgets(pc, 100, stdin);

len = strlen(pc)
printf("%s\n", pc);

if ( strcmp(pc, "quit") == 0 ) {
        printf("You entered 'quit'");
}
```

- The only difference between storage pointed to by `pc` and the storage defined by a `char` array is the time at which the storage becomes available to your program.

Initialization of Pointers

- A pointer must be given a value before it can be used properly. This happens either explicitly or through a function call, which returns a pointer, or by assignment from another pointer.
- The following sequence would cause a run time error, since the pointer has an uninitialized value.

```
char *pc;
gets(pc);
```

- A pointer can be used in the same context as an array, provided that the pointer has a meaningful value.
- Here are some legitimate pointer initializations.
  - Explicitly

```
char line[MAX];
char *pc, *p;
pc = line;
```

  - Through a function call

```
pc = malloc(100);
```

  - By assignment from another pointer

```
p = pc;
```

- Through the argument and parameter relationship

```
char line[100];          int strlen(char *s)
len = strlen(line)       { ...
```

Functions Returning a Pointer

- Another function that returns a pointer is the library function `fgets`, which can be used to get a line of input from the keyboard (`stdin`) or a file.

```
char * fgets(char * buffer, int size, FILE * stream);
```

- `fgets` returns a pointer to the character array that it is filling (or `NULL` on end of file or read error). `NULL` is typically used by functions returning a pointer when some problem or special condition needs to be indicated.

```
#define MAX 100

//  Read line-by-line and print lines with line
numbers

int main()
{
      char line[MAX];
      int  number = 1;

      while (fgets(line, MAX, stdin) != NULL) {
         printf("%4d%s\n", number++, line);
   }
}
```

- Since the value `NULL` is usually defined as zero,the while-loop could be coded simply as:

```
while (fgets(line, MAX, stdin))
```

- This code also uses the fact that `fgets` returns a pointer:

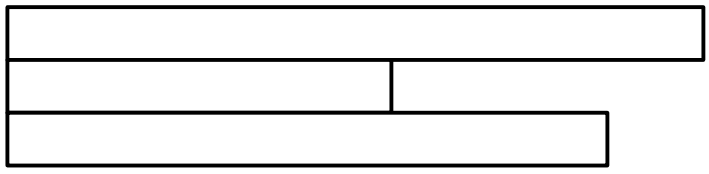
```
printf("Enter a number: ");
number = atoi(fgets(line, MAX, stdin));
```

Arrays of Character Pointers

- Character pointers often point to the first of many characters terminating with the `'\0'` character. Therefore, you can use an array of character pointers to handle a set of strings.

```
char *pc;              // a char pointer
char line[MAX];  // a char array
char *ptrs[MAX]; // an array of char pointers
```

- An array of character pointers is similar to a two-dimensional array, although more flexible. An array of character pointers occupies storage for the pointers, and ultimately these pointers will point to the strings.
- A two-dimensional character array occupies storage for the strings themselves. Two-dimensional arrays must be rectangular, so space is often wasted.
- An array of character pointers can point to strings of varying sizes ("ragged" array), and therefore, can be used more efficiently than a two-dimensional array.



Arrays of Character Pointers

- Consider the following definitions. The first can store `ROWS` many strings, each of maximum length `COLS - 1`. The second two stores pointers to `ROWS` many strings, each of which could be any length.

```
char lines[ROWS][COLS];
char *ptrs[ROWS];

printf("%s\n", lines[0]);
printf("%s\n", ptrs[0]);
```

  - In each `printf` above, the first string of the data structure is printed.

- However, the amount of memory used by each array is different.

```
sizeof(lines) =  ROWS * COLS
sizeof(ptrs)  =  ROWS * sizeof(char *)
```

- The next page shows an example of using an array of character pointers to store a set of strings.

Arrays of Character Pointers

pointer-array.c

```
 1. #include <stdio.h>
 2. #include <malloc.h>
 3.
 4. #define MAX 100     // number of lines
 5. #define LENGTH 80   // line length
 6.
 7. int main()
 8. {
 9.     char line[LENGTH + 1];
10.     char *ptrs[MAX];
11.     int len, i = 0, j;
12.
13.     while (fgets(line, LENGTH, stdin) != NULL) {
14.           if (i == MAX) {
15.              fprintf(stderr, "Sorry, array full\n");
16.              break;
17.           }
18.
19.           len = strlen(line);       //  strip off
20.           line[len - 1] = '\0';     //   newline
21.           len--;
22.
23.           ptrs[i] = malloc(len + 1);
24.           if (ptrs[i] == NULL) {
25.               fprintf(stderr, "Out of memory\n");
26.               exit(1);
27.           }
28.
29.           strcpy(ptrs[i++],line);
30.      }
31.
32.      for (j = 0; j < i; ++j) {
33.           printf("%s\n", ptrs[j]);
34.      }
35.
36.      for (j = 0; j < i; ++j) {
37.           free ptrs[j]);           // deallocate memory
38.      }
39. }
```

Command Line Arguments

- When a C program is executed from the command line, it is passed two parameters.
  - The first parameter is an integer representing the number of command line arguments provided to the program (including the command name).
  - The second parameter is a pointer to an array of pointers, each of which points to one of the command line arguments.
- If you wish to access the command line arguments, your main function must declare these parameters:

```
int main(int argc, char **argv)
```

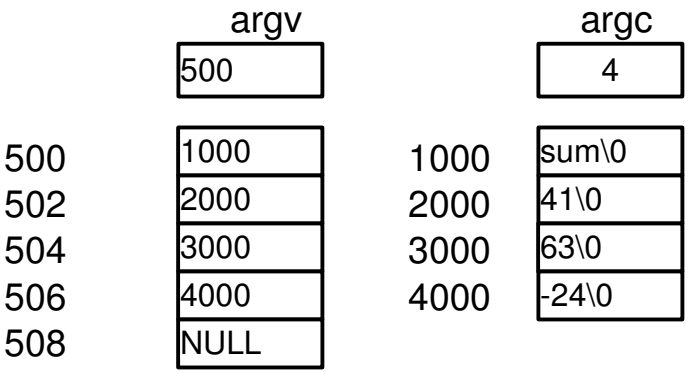
  - Of course you can name the two parameters any way you like, but `argc` (argument count) and `argv` (argument vector) are typically used for historical reasons.
- Be aware that all command line arguments come into your program as strings. If you want to treat them as numbers, they must be converted (with functions like `atoi` and `atof`).

Command Line Arguments

- The `argv` parameter is a pointer to the first of an array of pointers. If you dereference this pointer, you get a pointer (the first one in the array of pointers). If you use double indirection you get a character (the first character of the string pointed to by the first pointer).
- Suppose a C program is executed as follows:

```
sum 41 63 -24
```

- Below is the layout of memory for this command line:




```
 *argv  =  the address of "sum"    (char *)
**argv  =  the 's' in "sum"        (char)
```

Command Line Arguments

- This program sums the command line arguments.

```
int main(int argc, char **argv)
{
    int i, sum = 0;

    for(i = 1; i < argc; i++) {   // Skip program
name
        sum += atoi(argv[i]);
    }

    printf("sum is %d\n", sum);
}
```

- As in the code above, double indirection is not necessary but it is an idiom worth knowing, particularly if you need to access the individual characters of a command line argument.

`argv[n]` = the address of n^{th} argument
`argv[n][k]` = the k^{th} character of the n^{th} argument

- You can increment `argv` to advance it to the next pointer in the array to which it points. Therefore, another way of writing the code above is:

```
int main(int argc, char **argv)
{
    int i, sum = 0;

    ++argv;    // Skip program name

    for(i = 1; i < argc; i++) {
        sum += atoi(*argv++);
    }

    printf("sum is %d\n", sum);
}
```

Command Line Arguments

- This program includes a function that prints the command line arguments. The function relies on the fact that the array of pointers ends with the value `NULL`.

print-args.c

```
 1. #include <stdio.h>
 2.
 3. void printArgs(char **a);
 4.
 5. int main(int argc, char **argv)
 6. {
 7.     print(argv);
 8. }
 9.
10. void printArgs(char **a)
11. {
12.     ++a;          //  Skip program name
13.
14.     while(*a) {
15.         printf("%s\n",*a++);
16.     }
17. }
```

Command Line Arguments

- If you do need to access each character from each string on the command line, you could code as in either of the following two examples.

```
int main(int argc, char **argv)
{
     char *s;

     while(##argc > 0) {
         s = *++argv;
         while(*s) {

             /* process the character */

             s++;/* advance to next char */
         }
     }
}
```

```
int main(int argc, char **argv)
{
    char *s;
    int i;

    for (i = 1; i < argc; i++) {
        s = argv[i];
        while(*s) {

            /* process the character */

            s++;   /* advance to next char */
        }
    }
}
```

Command Line Arguments

- Many programs require a specific number of arguments on the command line. If the user does not comply, you can issue as error message. Here is an example to illustrate.

```
raise 2 10
```

- Here is a code snippet to process the command line shown above. Remember that `argc` is a count of the arguments and includes the command name.

```
int base, expo;

if (argc != 3 ) {
    fprintf(stderr, "Usage: message\n");
    exit(1);
}

base = atoi(argv[1]);
expo = atoi(argv[2]);
```

Practice with Pointers

- The following expressions pertain to the running of the hypothetical program `mcopy`.

```
mcopy input output
```

Expression	Value	Type
`argv`	`500`	`char **`
`*argv`	`1000`	`char *`
`argv[0]`	`1000`	`char *`
`*argv[0]`	`m`	`char`
`argv[0][0]`	`m`	`char`
`argv[0][1]`	`c`	`char`
`argv[1][0]`	`i`	`char`
`++argv`	`502`	`char **`
`*argv`	`2000`	`char *`
`**argv`	`i`	`char`
`*argv[0]`	`i`	`char`
`argv[0][0]`	`i`	`char`
`argv[0][1]`	`n`	`char`

- Note that `argv` can be treated as a two dimensional array. Recall that `argv[i][j]` is the j^{th} character of the string pointed to by the ith pointer in the array to which `argv` points.

Accessing Environment Variables

- The program below prints the value of an environment variable given as an argument on the command line.

envptr.c

```
 1. #include <stdio.h>
 2.
 3. int main(int argc, char** argv, char** envp)
 4. {
 5.     if(argc != 2) {
 6.        fprintf(stderr,
 7.          "usage: envptr <env_variable>\n");
 8.          exit(1);
 9.     }
10.
11.     while(*envp != NULL) {
12.          if (strstr(*envp, argv[1])) {
13.             printf("%s\n", *envp);
14.             exit(0);
15.          }
16.          else
17.             envp++;
18.     }
19.
20.     printf("%s not found\n", argv[1]);
21. }
```

- Sample execution:

```
$ envptr
usage: envptr <env_variable>

$ envptr PATH
PATH=/bin/usr:/user/bin:/user/michael/bin

$ envptr XYZ
XYZ not found
```

Accessing Environment Variables

- `envp` and `argv` are similar.
  - Both are of type `char **`.
  - The pointers being pointed to are **NULL** terminated.
- However, there is no analog to `argc`.
- Environment pointer is an important application tool.
- Programs can read the value of an environment variable for location of important items. Only exported environment variables are provided.

Function Pointers

- With the exception of the array type, the name of any variable is a reference to its value.
- The name of an array is a reference to its address.
- There is another type in C, which automatically generates an address. A function name without the function invocation operator is a reference to the address of the function.
- A function name can appear in argument lists.

```
int numbers[MAX], x = 25.0;
/*  the two calls below are very different
*/

/* The sqrt function is invoked   and it's value is
    sent to the stats function
*/

stats(numbers, MAX, sqrt(x));

/*  The address of the sqrt function
    is sent to the stats function
*/

stats(numbers, MAX, sqrt);
```

- In the latter case above, the third parameter must be capable of holding the address of a function. This gives rise to the concept of a pointer to a function.

Function Pointers

- Study the following.

```
/* int pointer   */
int *px;

/*  function returning int and taking
    no parameters
*/
int fun(void);

/*  function returning int pointer and
        taking no parameters
*/
int * morefun(void);

/*  pointer to a function taking no
        parameters and returning an int
*/
int (*pf)(void);
pf = fun;             /* see below */
```

- The last line above defines pf to be a pointer to a function.
  - A pointer such as this could be assigned the address of a function, which returns an int and takes no arguments.

```
int fun(void)
{
    int val;
    val = random(); /*random value */
    return val;
}
```

Function Pointers

- Suppose you have the following.

```
int square(int p) { return p * p; }
int cube(int p)   { return p * p * p; }
...
...
...
```

- The following code can be used to square each of the elements of an array.

```
void square_them(int *x, int howmany)
{
        int i;
        for (i = 0; i < howmany; i++)
            x[i] = square(x[i]);
}
```

- The following code can be used to cube each of the elements of an array.

```
void cube_them(int *x, int howmany)
{
        int i;
        for (i = 0; i < howmany; i++)
            x[i] = cube(x[i]);
}
```

- There would likely be more of these functions.

Function Pointers

- However, there is an easier way. `square_them` and `cube_them` are exactly the same except for the function called within them.
- If we could parameterize these functions, we could have one function take the place of each of these two (or however many there are).
- We will call this new function `calc`. It would be invoked as follows.

```
int x[5] = { 12, 23, 14, 25, 16};
calc(x, 5, square);
calc(x, 5, cube);
```

- In other words, we pass either the `square` or `cube` function as an argument to the `calc` function. For this to work correctly, the third parameter of `calc` must be a pointer to a function.

```
void calc(int *x,int amt, int (*ptf)(int))
{
        int i;
        for (i = 0; i < amount; i++)
            x[i] = (*ptf)(x[i]);
}
```

  - The expression `(*ptf)(x[i])` is an indirect invocation of the function being passed to `calc`.

Function Pointers

- This example uses an array of function pointers.

funs.c

```
 1. #define MAX 3
 2. #define SIZE 100
 3.
 4. int cube(int);
 5. int square(int);
 6. int times2(int);
 7.
 8. int getvalues(int * data, int size);
 9. void print(int * data, int size);
10. void compute(int * data, int size, int (*ptf)(int));
11.
12. main( ) {
13.     int j, numValues;
14.     int (*p[MAX])();
15.     int x[SIZE];
16.
17.     p[0] = cube;
18.     p[1] = square;
19.     p[2] = times2
20.
21.     for (j = 0; j < MAX; j++) {
22.       numValues = getvalues(x,SIZE); // FILL x
23.          print(x, numValues);              // PRINT x
24.          compute(x, numValues, p[i]);      // COMPUTE
25.          print(x, numValues);              // PRINT x
26.     }
27. }
28.
29. int cube(int x) {
30.     return x * x * x;
31. }
32.
33. int square(int x) {
34.     return x * x;
35. }
36.
37. int times2(int x) {
38.     return 2 * x;
39. }
```

Function Pointers

funs.c (continued)

```
40. int getvalues(int * data, int size) {
41.     int i;
42.
43.     for(i = 0; i < size; i++)
44.          data[i] = i + 1;
45.
46.     return i;
47. }
48.
49. void print(int * data, int size) {
50.     int i;
51.
52.     for(i = 0; i < size; i++)
53.          printf("%d ", data[i]);
54.
55.     printf("\n");
56. }
57.
58. void compute(int * data, int size, int(*ptf)(int)) {
59.     int i;
60.
61.     for(i = 0; i < size; i++)
62.          data[i] = ptf(data[i]);
63. }
```

Function Pointers

menu.c

```
 1. #include <stdio.h>
 2. #include <stdlib.h>
 3.
 4. #define MAX 100
 5.
 6. double pow(double, double);
 7. double atof(char *);
 8. double add (double, double);
 9. double mult(double, double);
10.
11. struct commands {
12.    char *name;
13.    double (*pf)();
14. } c[] = {"add", add, "mult", mult, "power", pow };
15.
16. #define SIZE sizeof(c)/ sizeof(struct commands)
17.
18. main( )
19. {
20.    char line[MAX];
21.    double a,b, ans;
22.    int i;
23.
24.    while(1) {
25.          printf("enter a command:\n");
26.          printf("add\nmult\npower\nquit\n => ");
27.          gets(line);
28.
29.          if(strcmp(line, "quit") == 0)
30.              break;
31.
32.          for (i = 0; i < SIZE; i++) {
33.              if(strcmp(c[i].name, line) == 0) {
34.                    printf("input first number ");
35.                    a = atof(gets(line));
36.                    printf("input second number ");
37.                    b = atof(gets(line));
38.                    ans = c[i].pf(a, b);
39.                    printf("ans is %f\n", ans);
40.                    break;
41.              }
42.        }
```

Function Pointers

menu.c (continued)

```
43.           if( i == SIZE )
44.               printf("%s not implemented\n",line);
45.
46.           printf("press return to continue\n");
47.           gets(line);
48.           system("clear") && system("cls");
49.       }
50. }
51.
52. double pow(double a, double b)
53. {
54.     int i;
55.     double temp = a;
56.
57.     for(i = 0; i < b - 1; i ++)
58.           temp *= a;
59.
60.     return temp;
61. }
62.
63. double add (double a, double b)
64. {
65.     return a + b;
66. }
67.
68. double mult(double a, double b)
69. {
70.     return a * b;
71. }
```

Function Pointers

sort.h

```
 1. #define NO_LINES 20
 2. #define ERROR #1
 3. #define LINE_LENGTH 100
 4.
 5. int numcmp(const char *, const char *);
 6. int revcmp(const char *, const char *);
 7. int revnum(const char *, const char *);
 8.
 9. void printlines(char **lines, int howmany);
10. int readlines(char **lines, int maximum);
11.
12. void sort(char **, int,
13.     int (*)(const char *, const char *));
```

- Functions used by the **sort** program

sort.c

```
 1. #include <stdio.h>
 2. #include <string.h>
 3. #include <stdlib.h>
 4.
 5. #include "sort.h"
 6.
 7. void printlines(char **ptrs,int lines)
 8. {
 9.     while(##lines >= 0)
10.         printf("%s\n",*ptrs++);
11. }
12.
13. void sort(char **p,int lines,int (*f)())
14. {
15.     int i, j;
16.     char *pc;
17.     for(i = 0;  i < lines # 1; i++)
18.         for(j = i + 1; j < lines; j++)
19.             if( f(p[i],p[j]) > 0 )
20.                 pc = p[i], p[i] = p[j], p[j] = pc;
21. }
```

Function Pointers

sort.c (continued)

```
22. int numcmp(const char *s, const char *t) {
23.    return(atof(s) # atof(t));
24. }
25.
26. int revnum(const char *s, const char *t) {
27.    return(atof(t) # atof(s));
28. }
29.
30. int revcmp(const char *s, const char *t) {
31.    return(strcmp(t, s));
32. }
```

Function Pointers

mainsort.c

```
 1. #include <stdio.h>
 2. #include <stdlib.h>
 3. #include <string.h>
 4. #include <malloc.h>
 5.
 6. #include "sort.h"
 7.
 8. main(int argc,char **argv) {
 9.     char *ptrs[NO_LINES], reply[10];
10.     int lines;
11.
12.     if(argc != 2) {
13.         printf("usage: sort #n numerical\n");
14.         printf("       sort #a alphabetical\n");
15.         printf("       sort #r reverse alpha\n");
16.         printf("       sort #rn reverse numeric\n");
17.         exit(3);
18.     }
19.
20.     if((lines = readlines(ptrs,NO_LINES)) == ERROR)
21.         printf("too many lines or no room\n");
22.         exit(1);
23.     }
24.
25.     if(strcmp(argv[1],"#n") == 0)
26.         sort(ptrs,lines,numcmp);
27.
28.     else if(strcmp(argv[1],"#a") == 0)
29.         sort(ptrs,lines,strcmp);
30.
31.     else if(strcmp(argv[1],"#r") == 0)
32.         sort(ptrs,lines,revcmp);
33.
34.     else if(strcmp(argv[1],"#rn") == 0)
35.         sort(ptrs,lines,revnum);
36.
37.     else {
38.         printf("incorrect argument\n");
39.         exit(4);
40.     }
41.     printlines(ptrs, lines);
42.
```

Function Pointers

mainsort.c (continued)

```
43.
44.    // deallocate dynamically allocated memory
45.
46.    for (i = 0; i < lines; i++) {
47.          free(ptrs[i]);
48.    }
49.
50. } // end of main
51.
52. int readlines(char **p,int limit)  {
53.    int n = 0;
54.    char *pc, line[LINE_LENGTH];
55.
56.    while(gets(line) != NULL) {
57.       if(n > limit)
58.          return(ERROR);
59.
60.       else if((pc = malloc(strlen(line) + 1))== NULL)
61.          return(ERROR);
62.
63.       else {
64.          strcpy(pc,line);
65.          p[n++] = pc;
66.       }
67.    }
68.    return(n);
69. }
```

Exercises

1. Write a program that takes three values from the command line: a beginning temperature (celsius), an ending temperature (celsius), and an increment value.

   The program should produce a temperature conversion chart. For example:

   ```
   myprog 0 30 10

   CELSIUS FAHRENHEIT
   0       32.0
   10      50.0
   20      68.0
   30      86.0
   ```

   The program should print appropriate error messages if the correct number of command line arguments is not supplied.

   - The formula for converting Celsius to Fahrenheit is:

     ```
     f = 1.8 * c + 32
     ```

2. Revisit the `extended.c` program in Chapter 3. Rewrite the main loop of that program so that the user specifies how many integers are to be input. Your program uses this value to allocate the right amount of memory using `malloc`.

3. Write a program which asks the user to input a string. Then, write a function called `lookup`, which determines if the input string was also a command line argument. Model your function after either one of the following:

   ```
   int lookup(char **argv, int count, char *str);
   char **lookup(char **argv,int count, char *str);
   ```

   - The session below demonstrates how the program should work:

Exercises

Exercise 3 (continued):

```
$ myprog these are some input strings

Enter a string: some
'some' was argument number 3

$ myprog these are some more strings

Enter a string: michael
'michael' was not an argument
```

4. Write a program which prints the sum of all the digit characters (not the number of them) on the command line. For example:

   program-name **24a 3b5 -23 c54 83**

   Output:
   39

5. The program `envptr` behaves incorrectly for any argument which is a substring of an actual environment variable. Why? Fix the problem.
6. Write a function which returns a pointer to a function returning an integer, and then use that function in a program.

This Page Intentionally Left Blank

Chapter 7:

Binary I/O and Random Access

A Database Application

- In this section, we are interested in functions used to read and write structures. The functions will be demonstrated by developing a typical financial application dealing with structures.
- The application uses the following header files:

employee.h

```
 1. #ifndef EMPLOYEE_H
 2. #define EMPLOYEE_H
 3.
 4. #define NAMESIZE 20
 5.
 6. struct employee
 7. {
 8.      char name[NAMESIZE + 1];
 9.      double pay;
10.      char dept;
11. };
12.
13. typedef struct employee WORKER, *PWKR;
14.
15. void fill(PWKR p);
16. void output(PWKR p);
17. void fill_emp(PWKR array, int size);
18. void print(PWKR array, int size);
19. PWKR lookup(char* name, PWKR array, int size);
20.
21. #endif
```

util.h

```
1. #ifndef UTIL_H
2. #define UTIL_H
3.
4. void stripnewline(char * str);
5.
6. #endif
```

A Database Application

dbfunc.h

```
 1. #ifndef DBFUNC_H
 2. #define DBFUNC_H
 3.
 4. #define FILENAMESIZE 14
 5.
 6. void print_db(void);
 7. void create_db(void);
 8. void retrieve_db(void);
 9.
10. #endif
```

- The `main` function (shown on next page) drives the rest of the program. It calls the `menu` function, which displays the following menu:

```
SELECT CHOICE

1) CREATE DATABASE
2) PRINT DATABASE
3) RETRIEVE
4) QUIT
==>
```

- Then `main` invokes the proper function to handle the user's selection.

A Database Application

mainprog.c

```
 1. #include <stdio.h>
 2. #include <stdlib.h>
 3. #include "employee.h"
 4. #include "dbfunc.h"
 5.
 6. #define MAXLINE   100
 7. #define CREATE_DB 1
 8. #define PRINT_DB  2
 9. #define RETRIEVE  3
10. #define QUIT      4
11.
12. int menu(void);
13.
14. main()
15. {
16.    char line[MAXLINE];
17.    int selection;
18.
19.    while(1)
20.    {
21.            selection = menu();
22.            switch(selection)
23.            {
24.              case CREATE_DB:
25.                    create_db();
26.                    break;
27.              case PRINT_DB:
28.                    print_db();
29.                    break;
30.              case RETRIEVE:
31.                    retrieve_db();
32.                    break;
33.              case QUIT:
34.                    exit(0);
35.              default:
36.                    printf("IMPOSSIBLE\n");
37.            }
38.
39.            printf("Press RETURN to continue\n");
40.            fgets(line, MAXLINE, stdin);
41.     }
42. }
```

The `menu` Function

- The `menu` function displays the menu and then gets a response from the user. After it verifies the response from the user, it sends the response back to the calling function.

mainprog.c (continued)

```
43. int menu(void)
44. {
45.     int choice;
46.     char line[MAXLINE + 1];
47.
48.     while(1)
49.     {
50.             printf("\n\n");
51.             printf("SELECT CHOICE\n\n");
52.             printf("1) CREATE DATABASE\n");
53.             printf("2) PRINT DATABASE\n");
54.             printf("3) RETRIEVE\n");
55.             printf("4) QUIT\n==>");
56.             choice = atoi(fgets(line, MAXLINE, stdin));
57.
58.             if(choice < 1 || choice > QUIT)
59.         {
60.               printf("ILLEGAL CHOICE\n");
61.               printf("PRESS RETURN TO CONTINUE\n");
62.               fgets(line, MAXLINE, stdin);
63.         }
64.         else
65.               return(choice);
66.     }
67. }
```

fwrite

- `fwrite` writes some data to the disk. The data written to the disk is in the exact format as it is in memory. This is called a **binary** write. The `fwrite` prototype is:

```
int fwrite(void *data, int size, int num, FILE *fp);
```

- `fwrite` requires four arguments:

`arg1`	Address of the data to be written
`arg2`	Size of each data item to be written
`arg3`	Number of data items to be written
`arg4`	`FILE` pointer

- `fwrite` returns the number of objects written. Here are some examples of using `fwrite`.

```
int i, n[10];
WORKER x, a[10];

/* write an int       */
fwrite(&i, sizeof(int), 1, fp);

/* write 10 ints      */
fwrite(n, sizeof(int), 10, fp);

/* write a struct     */
fwrite(&x, sizeof(WORKER), 1, fp);

/* write 10 structs   */
fwrite(a, sizeof(WORKER), 10, fp);

/* write second half of array     */
fwrite(a + 5, sizeof(WORKER), 5, fp);
```

The `create_db` Function

- The `create_db` method asks the user for a file name, creates the file, and then asks the user to input some records.
  - Notice the "wb" mode. This allows the binary write (and is not necessary on UNIX / Linux).

dbfunc.c

```
 1. #include <stdio.h>
 2. #include <string.h>
 3. #include "employee.h"
 4. #include "dbfunc.h"
 5. #include "util.h"
 6.
 7. void create_db()
 8. {
 9.     WORKER person;
10.      FILE *fp;
11.      char fname[FILENAMESIZE + 1], line[101];
12.      int number, i;
13.
14.      printf("Enter filename to write to: ");
15.     fgets(fname, FILENAMESIZE, stdin);
16.     stripnewline(fname);
17.      if((fp = fopen(fname, "wb")) == NULL) {
18.            printf("Can't open %s\n", fname);
19.            return;
20.      }
21.
22.      printf("How many records to input? ");
23.   number = atoi(fgets(line, 100, stdin));
24.
25.   for (i = 0; i < number; i++)
26.   {
27.            fill(&person);
28.            fwrite(&person, sizeof(WORKER), 1, fp);
29.      }
30.
31.     fclose(fp);
32. }
```

fread

- The `fread` function is the companion to `fwrite`. It has the exact same interface. Use this function to read data that has been written by `fwrite`. The `fread` prototype is:

```
int fread(void *data, int size, int num, FILE *fp);
```

- `fread` requires four arguments:

`arg1`	Address where data will be stored
`arg2`	Size of each data item to be read
`arg3`	Number of data items to be read
`arg4`	`FILE` pointer

- `fread` returns the number of objects read. When this value is zero, you have reached the end of the file.
- Here is a code segment which reads `SIZE` records at a time:

```
while(num > 0 )
{
      for (i = 0; i < num; i++)
   {
         /*  process each record */
   }
    /*  read another "chunk"  */
     num = fread(bank, sizeof(WORKER), SIZE, fp);
}
```

The `print_db` Function

- The `print_db` function gets a file name from the user and displays that file on the standard output. If the user gives a bad file name, the function returns.

dbfunc.c (continued)

```
33. void print_db()
34. {
35.      WORKER person;
36.        FILE *fp;
37.     char fname[FILENAMESIZE + 1];
38.
39.     printf("Enter filename: ");
40.     fgets(fname, FILENAMESIZE, stdin);
41.     stripnewline(fname);
42.
43.      if((fp = fopen(fname,"rb")) == NULL) {
44.            printf("Can't open %s\n", fname);
45.            return;
46.      }
47.
48.        while(fread(&person, sizeof(WORKER), 1, fp) > 0 )
49.            output(&person);
50.
51.     fclose(fp);
52. }
```

- Both `create_db` and `print_db` use the following form of the return statement.

```
return; /* premature exit from void function */
```

fseek

- Files are often read and written sequentially. The operating system keeps a pointer to the next record to be read or written. You can control this pointer with the `fseek` function. The prototype for `fseek` is:

```
int fseek(FILE *fp, long offset, int origin);
```

- `fseek` positions the file at `offset` many bytes from `origin`. The `origin` is given using a set of values defined in `stdio.h`:

```
#define SEEK_SET  0  /* From beginning   */
#define SEEK_CUR  1  /* From current pos */
#define SEEK_END  2  /* From end of file */
```

- Here are some examples of `fseek`:

```
long int n = 10;
long int size = sizeof(WORKER);

/* seek to beginning of file  */
fseek(fp, 0, SEEK_SET);

/* seek to end of file */
fseek(fp, 0, SEEK_END);

/* back up 1 record */
fseek(fp, -size, SEEK_CUR);

/* skip the first n records */
fseek(fp, size * n, SEEK_SET);
```

fseek

- `fseek` can be used to update a record.
  - Note the use of the `fflush` function, which causes immediate output to the file rather than having the output buffered.

```
WORKER person;
long int size = sizeof(WORKER);

while((fread(&person, size, 1, fp)) > 0)
{
    if(strcmp(pname, person.name) == 0)
    {
        /*  modify 'person' structure  */

        fseek(fp, -size, 1);
        fwrite(&person, sizeof(WORKER), 1, fp);
        fflush(fp);
    }
}
```

- `fseek` can be used to determine the number of records in a file.

```
long int size, recs;
char filename[] = "somefile";

fp = fopen(filename, "r");

fseek(fp, 0L, 2);              // Seek to end of file
size = ftell(fp);
recs = size / sizeof(WORKER);

printf("File %s is %ld bytes\n", filename, size);
printf("File %s has %ld records\n", filename, recs);
```

The `retrieve_db` Function

- `retrieve_db` is similar to `print_db` except that only selected records are printed.
  - Notice the `fseek` back to the beginning of the file to search the file again for a different name.

dbfunc.c (continued)

```
53. void retrieve_db()
54. {
55.      WORKER w;
56.      FILE *fp;
57.      char fname[FILENAMESIZE + 1],
58.     char pname[NAMESIZE + 2];
59.
60.   printf("Enter file name: ");
61.     fgets(fname, FILENAMESIZE, stdin);
62.     stripnewline(fname);
63.      if ((fp = fopen(fname, "r")) == NULL)
64.      {
65.            printf("Can't open %s\n",fname);
66.            return;
67.      }
68.
69.      while(1)
70.      {
71.            printf("Which name? ('quit' to exit) ");
72.            fgets(pname, NAMESIZE, stdin);
73.            stripnewline(pname);
74.            if (strcmp(pname, "quit") == 0)
75.                break;
76.
77.            while((fread(&w, sizeof(WORKER),1, fp)) > 0)
78.            {
79.                if(strcmp(pname, w.name) == 0)
80.                    output(&w);
81.            }
82.            fseek(fp,0L,0);
83.      }
84.      fclose(fp);
85. }
```

The Utility Functions

- The source file `util.c` holds common functions that might be needed in a variety of applications.
- Currently, there is just one function: `stripnewline`. This function behaves like the UNIX `chomp` utility - i.e., it removes the last character of a string, but only if that character is the newline.
- The program is designed to work so that it doesn't matter whether or not the input line ends in the newline character. Recall that:
  - The `gets` function discards the newline.
  - The `fgets` function retains the newline.

util.c

```
 1. #include <string.h>
 2.
 3. void stripnewline(char * str)
 4. {
 5.      int len = strlen(str);
 6.
 7.      if (str[len - 1] == '\n')
 8.     {
 9.             str[len - 1] = '\0';
10.      }
11. }
```

Exercises

1. Extend the database application program by providing a menu choice, which displays the number of records in the data file.
2. Add another menu choice so that the user can display any record by specifying the record number.
3. Write a program, which exchanges the first and last records of a file consisting of records of the WORKER type.

Chapter 8:

Designing Data Types

Steps in Creating Data Types

- New data types are defined using the `struct` keyword.
- The **typedef** facility merely gives a new name for an **existing** data type.
- Creating new data types is invaluable toward designing solutions that closely model the problem to be solved.
- To design a new data type:
  - Create a description for the data type structure description.
    - Typedef
  - Create the operations for this data type.
    - Function Prototypes
  - Write the implementation for the operations.
    - Function Definitions

Rationale for a New Data Type

- Suppose we wish to create a **fraction** data type. Our goal is to use fractions with the same ease as we use the built in types.
- C supports fundamental types such as:
  - char
  - int
  - double
    - Easy to write programs using these data types.
- We would like to use the same constructs for created data types as we do for the built in types.
- For example, we would like to define, add, and multiply **fractions** just as we do **ints**.
  - We can do all of this but we have to trade operators for functions.
  - We can simulate the **+** with a function.

The File `fraction.h`

- The file **`fraction.h`** would consist of the pieces shown here.
- Create it. In other words, decide on a representation.

```
struct fraction {
    int n;
    int d;
};
```

  - Note: Other representations are possible.

- Create new name(s) for the new type.

```
typedef struct fraction FRACTION, *FPTR;
```

- Specify the operations for the new type.

```
FRACTION create(int numerator, int denominator);
FRACTION input(void);
void print(FPTR);
int gcd(int first_dividend, int second_dividend);
FRACTION add(FPTR, FPTR);
FRACTION mult(FPTR, FPTR);
FRACTION divide(FPTR, FPTR);
FRACTION subtract(FPTR, FPTR);
etc.
```

Operations on the Fraction Data Types

```
int a,b,c;              // DEFINE INTs
FRACTION x,y,z;         // FRACTIONs

int fun()               // FUNCTION RETURNS INT
FRACTION fun2();        // RETURNING FRACTION

int *p1;                // POINTER TO INT
FRACTION *p2;           // TO FRACTION

a = b + c;              // ADD INTEGERS
x = y + z;              // ADD FRACTIONS
```

Implementation of the Functions

- Each of the functions should now be implemented.

fraction.c

```
 1. #include <stdio.h>
 2. #include <assert.h>
 3.
 4. #include "fraction.h"
 5.
 6. FRACTION input()
 7. {
 8.     FRACTION p;
 9.    printf("input num then denom (n/d) ");
10.
11.     //  "%d/%d"  format => allows
12.     //  input of the form  n/d
13.
14.     scanf("%d/%d", &p.n, &p.d);
15.     while( getchar() != '\n');      // flush the
16.                                     //  newline
17.     return(p);
18. }
19.
20. FRACTION create(int numer, int denom)
21. {
22.     FRACTION p;
23.     p.n = numer;
24.     p.d = denom;
25.     return(p);
26. }
```

Implementation of the Functions

fraction.c (continued)

```
27. void print(FPTR p)
28. {
29.     int temp;
30.
31.     assert(p -> d != 0 );        // div by 0
32.     temp = gcd(p->n, p->d);      // reduce
33.     assert(temp != 0);           // sanity check
34.
35.     p -> n = p -> n / temp;
36.     p -> d = p -> d / temp;
37.     if ( 1 == p -> d)            // easy reading
38.          printf("%d\n", p -> n);
39.     else if ( 0 == p -> n)       // easy reading
40.          printf("0\n");
41.     else
42.          printf("%d/%d\n", p -> n, p -> d);
43. }
44.
45. FRACTION add(FPTR f, FPTR s)
46. {
47.     FRACTION p;
48.
49.     p.n = f -> n * s -> d + f -> d * s -> n  ;
50.     p.d = f -> d * s -> d;
51.     return(p);
52. }
```

Implementation of the Functions

fraction.c (continued)

```
53. int gcd(int top, int bot)
54. {
55.     int quot, rem;
56.
57.     quot = top / bot;
58.     rem = top % bot;
59.     while(rem != 0)
60.     {
61.          top = bot;
62.          bot = rem;
63.          quot = top / bot;
64.          rem = top % bot;
65.     }
66.     return bot;
67. }
```

Example Program Using Fractions

- An entire program using the `FRACTION` type
  - Fill an array with fractions and compute their sum.

mainfrac.c

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

#define MAX 5

main()
{
        int i;
        FRACTION array[MAX];
        FRACTION s;

        s = create(0,1);

        printf("Enter %d fractions\n", MAX);
        for (i = 0; i < MAX; i++) {
             printf("input fraction # %d ", i + 1);
             array[i] = input();
        }

        for ( i = 0; i < MAX; i++)
             s = add(&s, &array[i]);

        print(&s);
}
```

Applications with Fractions

- Any applications with fractions would include the header file **`fraction.h`**. Fractions could then be:
  - defined with:
    - FRACTION a,b,c;
  - initialized with:
    - a = create(2,3);
    - b = create(4,5);
  - added with:
    - c = add(&a, &b);
- Complex calculations could proceed as:

```
c = (a + b) * (a / b);
```

```
FRACTION a, b, c;
c = mult(add(&a, &b), divide(&a, &b));
```

  - Note that **add** must return an **fptr** rather than a **fraction** to satisfy **mults** arguments.

Set Notation Examples

- Mathematicians use the following set notation.
  - A = { 0, 3, 5, 2, 4 } B = { 3, 4 , 6 , 1 }

- Set intersection **^**
  - A ^ B = { 3, 4 }

- Set union **u**
  - A u B = { 0, 1, 2, 3, 4, 5, 6 }

- Set difference **-**
  - A - B = { 0, 5, 2 }

- Compliment of a set **~**

- With respect to (say the set of digits)
  - ~A = { 1, 6, 7, 8, 9 }

Creating the Set Type

- Sets are collections of unordered objects.
- Many operations are defined on sets including:
  - **Intersection**: those elements in **A and in B**
  - **Union**: those elements in **A or in B**
  - **Difference**: those elements **in A and not in B**
  - **Complement** of a set: those **not in the set**
    - Complement of a set must be with respect to a universal set (say the set of all digits).

Set Representation Example

- The following pieces would be placed in **`set.h`**.
- Representation

```
#define SIZE 100
struct set {
        int array[SIZE];
        int howmany;
};
```

- Create the new names for the type.

```
typedef struct set SET, *SETP;
```

- Create operations for the new type.

```
SETP create(void);
void add(int, SETP);
void print(SETP);
SETP setunion(SETP, SETP);
```

Set Representation

- There are many representations of sets.
  - Abstract sets have no bound.
  - Good candidate for dynamic representation
- Instead, we choose the following.
  - An array
  - A particular size
- Following the method of the **fraction** data type, we use **typedef** to create some new names.
- Next, we choose the operations to be implemented.

Set Function Implementations

set.c

```
 1. #include <stdio.h>
 2. #include <stdlib.h>
 3. #include <malloc.h>
 4.
 5. #include "set.h"
 6.
 7. SETP create()
 8. {
 9.        SETP temp;
10.
11.        temp = (SETP) malloc(sizeof(SET));
12.        if(temp == NULL) {
13.             printf("malloc: no more room\n");
14.             exit(1);
15.        }
16.        temp #> howmany = 0;
17.        return(temp);
18. }
19.
20. void add(int new, SETP p)
21. {
22.        if( p #> howmany == SIZE ) {
23.             printf("set overflow\n");
24.             exit(2);
25.        }
26.        p #> array[p #> howmany++] = new;
27. }
28.
29. void print(SETP p)
30. {
31.        int i;
32.
33.        for ( i = 0; i < p #> howmany; i++)
34.             printf("%d\n", p #> array[i]);
35. }
```

Set Function Implementations

set.c (continued)

```
36. SETP setunion(SETP a, SETP b)
37. {
38.         SETP c;
39.         int i, j;
40.
41.         c = create();
42.         for (i = 0; i < a #> howmany; i++)
43.              add(a #> array[i], c);
44.         for (i = 0; i < b #> howmany; i++)  {
45.              for (j = 0; j < a #> howmany; j++)
46.                   if(b #> array[i] == a #> array[j])
47.                        break;
48.              if (j == a #> howmany)
49.                   add(b #> array[i], c);
50.      }
51.      return(c);
52. }
```

A Program That Uses the Set Data Type

mainset.c

```
53. #include "set.h"
54.
55. main()
56. {
57.         SETP c, a, b;
58.
59.         a = create();
60.         b = create();
61.         add(10, b);          // ADD ELEMENT 10 TO b
62.         add(5, b);
63.         add(5, a);           // ADD ELEMENT 5 TO a
64.         add(6, a);
65.         print(a);
66.         print(b);
67.
68.         c = setunion(a,b);   // UNION OF a AND b
69.         print(c);
70.
71.         free(a);
72.         free(b);
73.         free(c);
74. }
```

Exercises

1. Implement the fraction **add** function with the following function prototype.

   ```
   FPTR add(FPTR, FPTR);
   ```

   There is a starter file for this exercise in the `starters` directory.

2. Create the type **COMPLEX** (for complex numbers) and provide operations such as **init**, **add**, and **print**.

3. Implement **set** intersection. There is a starter file for this exercise in the `starters` directory.

Chapter 9:

Linked Lists

What are Lists?

- An element of a list is usually defined as a structure.
  - A passenger

```
struct passenger {
    char name[20];
    int flight_no;
    char seat[4];
};
```

  - A job in an operating system

```
struct job {
    int owner;
    int priority;
    char *files[20];
};
```

  - A window

```
struct window {
    int x_upper_left;
    int y_upper_left;
    int x_lower_right;
    int y_lower_right;
};
```

What are Lists?

- A list is a collection of (usually) like objects.
  - passengers on an airline
  - jobs in an operating system
  - windows on a display
- Lists are usually dynamic.
  - The number of elements in the list varies with time.
  - There is no upper limit on the size of the list.
- Common list operations include:
  - adding an element
  - inserting an element
  - deleting an element
  - printing the list
  - combining two lists

Lists as Arrays

- A list is a dynamic data structure. An array is **fixed**.
- This contradiction leads to inefficiencies in:
  - Adding to the list
    - what about when there is no more room?
  - Deleting from a list
    - moving too much data
  - Combining two lists
    - not room enough in either list

Lists as Arrays

- An array can be used to represent a list.
  - An array is a fixed size data type.
  - Size is based on a worst-case scenario.
  - The number of elements in the list would be kept in a separate variable.
- Array representation could lead to inefficiencies.
  - Adding an element
    - Since an array is a fixed data structure, there would be no way of **extending** it.
  - Inserting an element
    - All elements below the inserted one would need to be pushed down one element.
  - Deleting an element
    - Each element needs to be moved up a position, or, the position of the deleted one could be marked with a special value.
  - Combining two lists
    - The sum of the number of elements from the two lists could be larger than the capacity for either array.
- A better representation than an array is a data structure, which is allocated only when it is needed.
  - The need is usually signaled by a user request to the program.

Benefits of Linked Lists

- Linking elements is a dynamic way of building lists.
- The problem of **fixed size** disappears.
- Deleting an element becomes a matter of pointer manipulation.
- Inserting an element also is a pointer manipulation problem.
- Combining two lists need not worry about size restriction.

A List of Linked Elements

- Create the storage for an element when it is needed.
  - Describe a template for any list element.
  - Allocate as needed - **malloc**.
- Leads to many allocations at various locations
  - Provide an extra structure member, a pointer.
  - Link each new allocation to previous ones.
    - Insert append in order
- Inserting or appending an element
  - The problem of extending the array size disappears.
- Deleting an element
  - There is no inefficiency involved as with an array.
- Combining two lists
  - There is no concern about combined sizes.
  - Modify one pointer.

Defining the List Data Type

- The header file **`list.h`** consists of:

```
#defines
struct definition
typedefs
prototypes
```

- Create the new type.

```
#define NAMESIZE 20
#include <stdio.h>
struct passenger {
    char name[NAMESIZE];
    int f_number;
    struct passenger *link;
};
```

- Give it a name.

```
typedef struct passenger PASS, *PP;
typedef int BOOLEAN;
```

- Define the operations.

```
void initialize(PP *);    // INITIALIZE THE LIST

BOOLEAN is_empty(PP)      // IS THE LIST EMPTY?

PP insert(PP, PP);        // ADD TO FRONT OF LIST

PP append(PP, PP);        // ADD TO END OF LIST

PP create(char *, int);   // CREATE, FILL ELEMENT

void print(PP);           //   PRINT THE LIST
```

The List Data Type

- Think of a list as a new data type.
  - Define the representation.
  - Define the operations.
  - Implement the functions.
- A list will be represented as:
  - the data
  - a pointer to the rest of the list.
- A minimum set of operations on a list would be:

```
initialize        // INITIALIZE THE LIST

insert            // ADD TO FRONT OF LIST

append            // ADD TO END OF LIST

create            // CREATE AND FILL ELEMENT

print             // PRINT THE LIST
```

- The file `list.h` should contain all the above information.

Implementations of List Functions

- Any functions using the **list** type must include **`list.h`**.
- The `initialize` function would be:

```
#include "list.h"
void initialize(PP *p)
{
     *p = NULL;
}
```

- `is_empty` determines if the list is empty.

```
BOOLEAN is_empty(PP p)
{
     return( p == NULL );
}
```

- `print` prints the list.

```
void print(PP p)
{
     while (p != NULL) {
          printf("%s %d\n", p #>name, p #>f_number);
          p = p #> link;
     }
}
```

Implementation of List Functions

- The `create` function

```
PP create(char *name, int number)
{
     PP p;
     p = (PP) malloc(sizeof(PASS));
     if( p == NULL ){
          printf("malloc: no more room\n");
         return(NULL);
     }
     p #> f_number = number;
     strcpy(p #> name, name);
     p #> link = NULL;
     //FOR TWO WAY LINKED LISTS
     //  p -> blink = NULL;    LATER
     //       IN THE CHAPTER
     return(p);
}
```

- The `insert` function

```
PP insert(PP data, PP p)
{
     data #> link = p;
     return(data);
}
```

Implementation of List Functions

- The `append` function

```
PP append(PP data, PP p)
{
    PP temp = p;
    PP prev;
    if(is_empty(p)) {
        p = insert(data, p);
        return(p);
    }
    while(p != NULL) {
        prev = p;
        p = p #> link;
    }
    //    prev POINTS TO LAST ELEMENT
    prev #> link = data;
    return(temp);
}
```

A Simple Program With a List

listmain.c

```
 1. #include "list.h"
 2.
 3. main( )
 4. {
 5.     PP list, node;
 6.
 7.     initialize(&list);
 8.
 9.     node = create("mike",100);
10.     list = insert(node,list);
11.
12.     node = create("tom",1);
13.     list = append(node,list);
14.
15.     if(! is_empty(list))
16.            print(list);
17. }
```

Other Types of Lists

- The list from the previous section is called a one way linked list.
- One way lists are generally better than arrays.
  - Many deletions and additions
  - Fairly simple to manage
- In the one way list presented earlier, we had no concern for order.
  - Insert at the beginning
  - Append at the end
  - An ordered list adds elements in order based on a key field of the record.
- Other types of lists give other benefits.

Other Types of Lists

- A one way list has certain deficiencies.
  - Can't visit any element from any other element
  - A circular list solves this problem.
  - Always start searching from current position
- Some list problems require additional pointers to provide additional efficiency.
  - May have need for doubly-linked list
  - An editor with a linked list of lines

```
print 1,10   //  PRINT LINES 1 - 10

print ., -5  //  PRINT PREVIOUS FIVE LINES
```

- May have a list where each element itself contains a head pointer for another list (a network)
  - Linked airline flights with linked passengers
  - Linked words in a file with linked line number, (i.e, a cross reference)

Ordered Lists

- Below is an example of an ordered list.

```
#include "ordered.h"

main()
{
PP list, node;
initialize(&list);
     node = create("mike", 100);
list = order(node, list);
node = create("zeke", 100);
list = order(node, list);
node = create("allen", 100);
list = order(node, list);
node = create("mikey", 100);
list = order(node, list);
print(list);
}

PP order(PP data, PP p)
{
PP prev;
PP save = p;

if(is_empty(p) ||
       (strcmp(data #> name, p #> name) < 0 )) {
         data #> link = p;
         return(data);
}
while((p != NULL) &&
           (strcmp(data #> name, p #> name) > 0)) {
          prev = p;
          p = p #> link;
}
data #> link = prev #> link;
prev #> link = data;
return(save);
}
```

Ordered Lists

- Same as previous linked list except insertion is a function of a key field (name for example)
- The function order replaces the combination of insert and append.
- Elements are now in order.

The `rand` Function

- Random numbers will be needed in many of the following examples.
- The `rand` function yields random numbers in the range:

```
0 - 32767        (FOR 16 BIT INTEGERS)
```

- The `rand` function yields random numbers in the range:

```
0 - (2 ** WORDSIZE - 1)
```

```
#define HOW_MANY 100
main( )
{
int i;
for (i = 0; i < HOW_MANY; i++)
        printf("%d\n", rand( ));
}
```

- Use the % operator to scale a random number to the range 0 - (n - 1).

```
rand( ) % n
```

- To scale a random number to the range BEGIN to END.

```
(rand() % (1 + END - BEGIN)) + BEGIN
```

Circular Lists

- To demonstrate the **circular** list, devise the following problem.
  - Consider a set of integers in the range **0** to **n**.
  - Select a random number i in the range (**1 - n**).
  - Eliminate the **i**th integer from the beginning, then the 2ith integer etc., until only one integer remains.
  - For example, if **i = 3**, eliminate 2,5,8 etc.
  - For any integer > **I**, wrap around.
  - Which number remains for various values of **i**?

Circular Lists

circular.c

```
 1. #include <stdio.h>
 2. #include <string.h>
 3. #include <malloc.h>
 4.
 5. #include "list.h"
 6.
 7. void initialize(PP *p)
 8. {
 9.     *p = NULL;
10. }
11.
12. BOOLEAN is_empty(PP p)
13. {
14.     return( p == NULL ? 1 : 0 );
15. }
16.
17. void print(PP p)
18. {
19.     while (p != NULL) {
20.         printf("%s %d\n", p -> name, p -> f_number);
21.         p = p -> link;
22.     }
23. }
24.
25. PP create(char *name, int number)
26. {
27.     PP p;
28.
29.     p = (PP) malloc(sizeof(PASS));
30.     if( p == NULL ){
31.          printf("malloc: no more room\n");
32.          return(NULL);
33.     }
34.
35.     p -> f_number = number;
36.     strcpy(p -> name, name);
37.     p -> link = NULL;
38.     return(p);
39. }
```

Circular Lists

circular.c (continued)

```
40. PP insert(PP data, PP p)
41. {
42.     data -> link = p;
43.     return(data);
44. }
45.
46. PP append(PP data, PP p)
47. {
48.
49.     PP temp = p;
50.     PP prev;
51.     if(is_empty(p)) {
52.         p = insert(data, p);
53.         return(p);
54.     }
55.     while(p != NULL) {
56.         prev = p;
57.         p = p -> link;
58.     }
59.     //    prev points to last element
60.     prev -> link = data;
61.     return(temp);
62. }
```

circularmain.c

```
 1. #include <stdio.h>
 2. #include <malloc.h>
 3.
 4. #include "list.h"
 5.
 6. #define NUMBER  22
 7.
 8. main()
 9. {
10.     char map[NUMBER + 1];
11.     int i, count;
12.     PP list, node, p, prev, save;
13.     initialize(&list);
```

Circular Lists

circularmain.c (continued)

```
14.     for ( i = 0; i < NUMBER; i++) {
15.         node = create("tom",i);
16.         list = append(node,list);
17.         map[i] = 'X';
18.     }
19.     map[NUMBER] = '\0';
20.
21.     p = list;
22.     for (i = 0; i < NUMBER - 1; i++)
23.         p = p -> link;
24.
25.     p -> link = list;
26.     p = list;
27.
28.     for (count = 0; count < NUMBER - 1; count++) {
29.         for(i = 0; i < 2; i++) {
30.             prev = p;
31.             p = p -> link;
32.         }
33.         save = p;
34.         prev -> link = p -> link;
35.
36.         map[p -> f_number] = '_';
37.         printf("%s\n",map);
38.         free(p);
39.         p = prev -> link;
40.     }
41. }
```

- `map` is used to display the results.
- `free` returns what `malloc` took.
- Always give `free` a pointer obtained by `malloc`

Circular Lists

- There are some basic differences between a circular list and a one directional list.
  - There is no first element.
  - There is no end of the list.
- There will be a current element.
  - This is the starting point for the next list operation.
- Each element points to the next one.
- Circular lists are used heavily in memory management schemes.

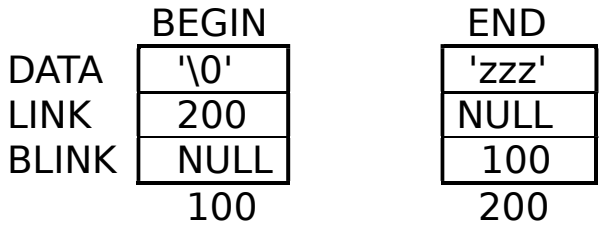
Two Way Lists

- A two_way list structure needs two pointers. The header file `list.h` needs a slight revision.

twoway.h

```
 1. #define NAMESIZE 20
 2.
 3. struct passenger {
 4.   char name[NAMESIZE];
 5.   int f_number;
 6.   struct passenger *link;
 7.   struct passenger *blink;
 8. };
 9.
10. typedef struct passenger PASS, *PP;
11. typedef int BOOLEAN;
12.
13. BOOLEAN is_empty(PP);     //   IS THE LIST EMPTY?
14. PP order(PP, PP);         //   ADD IN ORDER
15. PP create(char *, int);   //   CREATE AND FILL ELEMENT
16.
17. void fprint(PP);          //   PRINT THE LIST FORWARD
18. void bprint(PP);          //   PRINT THE LIST BACKWARD
```

- Two dummy nodes make the code more efficient.



Two Way Lists

- Two way lists allow processing in either direction.
  - Can cut down access time
  - Print from jones to smith
  - Print smith and preceding 4
  - Very often two way lists are also circular.
- The list has two pointers.
  - Forward pointer link
  - Backward pointer blink
- There are two print routines - forward, backward.
- To make the order function easier and more efficient, two dummy nodes are established in such a way that any new nodes necessarily fits between them.

Two Way Lists

twoway.c

```
 1. #include <stdio.h>
 2. #include <string.h>
 3. #include <malloc.h>
 4.
 5. #include "twoway.h"
 6.
 7. void fprint(PP p)
 8. {
 9.     printf("FORWARD\n");
10.     while (p -> link != NULL) {
11.         printf("%s %d\n", p -> name, p -> f_number);
12.         p = p -> link;
13.     }
14. }
15.
16. void bprint(PP p)
17. {
18.     printf("BACKWARD\n");
19.     while (p -> blink != NULL) {
20.         printf("%s %d\n", p -> name, p -> f_number);
21.         p = p -> blink;
22.     }
23. }
24.
25. PP create(char *name, int number)
26. {
27.     PP p;
28.     p = (PP) malloc(sizeof(PASS));
29.     if( p == NULL ){
30.          printf("malloc: no more room\n");
31.          return(NULL);
32.     }
33.     p -> f_number = number;
34.     strcpy(p -> name, name);
35.     p -> link = NULL;
36.     p -> blink = NULL;
37.     return(p);
38. }
```

Two Way Lists

twoway.c

```
39. PP order(PP data, PP p)
40. {
41.     PP prev;
42.     PP save = p;
43.     while(strcmp(data -> name, p -> name) >= 0)  {
44.         prev = p;
45.         p = p -> link;
46.     }
47.     data -> link =  prev -> link;
48.     prev -> link = data;
49.     data -> blink = p -> blink;
50.     p -> blink = data;
51.     return(save);
52. }
```

twowaymain.c

```
 1. #include <stdio.h>
 2. #include <stdlib.h>
 3. #include <string.h>
 4. #include <malloc.h>
 5.
 6. #include "twoway.h"
 7.
 8. main()
 9. {
10.     int i;
11.     PP begin, end, node, list;
12.
13.     begin = (PP) malloc(sizeof(PASS));
14.     end = (PP) malloc(sizeof(PASS));
15.     if(begin == NULL || end == NULL) {
16.         printf("malloc: no room\n");
17.         exit(1);
18.     }
19.
20.     strcpy(begin -> name, "\0");
21.     strcpy(end -> name, "z");
```

Two Way Lists

twowaymain.c (continued)

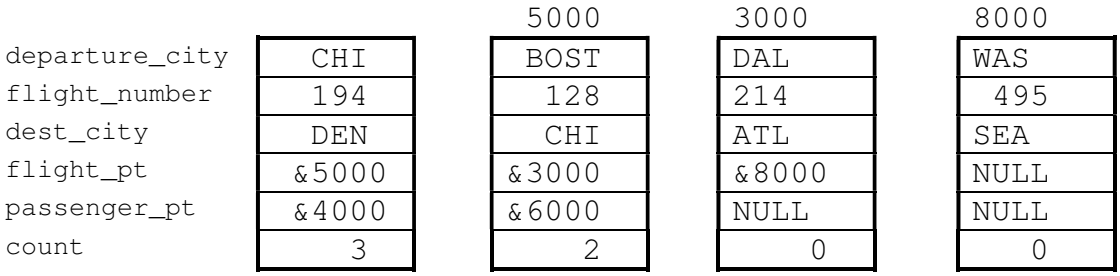
```
22.     begin -> link = end;
23.     end -> link = NULL;
24.
25.     begin -> f_number = 100;
26.     end -> f_number = 200;
27.
28.     end -> blink = begin;
29.     begin -> blink = NULL;
30.
31.     node = create("mike", 100);
32.     begin = order(node, begin);
33.     node = create("warren", 100);
34.     begin = order(node, begin);
35.     node = create("allen", 100);
36.     begin = order(node, begin);
37.     node = create("alice", 100);
38.     begin = order(node, begin);
39.     node = create("joe", 100);
40.     begin = order(node, begin);
41.     fprint(begin -> link);
42.     bprint(end -> blink);
43. }
```

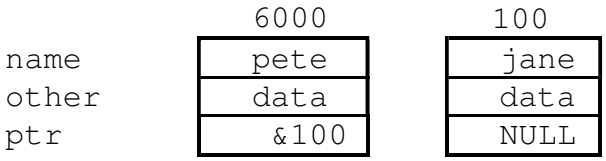
Nested Lists

```
struct flights {
char departure_city[20];
int flight_number;
char destination city[20];
struct flights *flight_pt;
struct passenger *passenger_pt;
};

struct passenger {
char name[20];
char other[100];
struct passenger *ptr;
};

typedef struct flights FLIGHT, *FPTR;
typedef struct passenger PASS, *PPTR;
```

Nested Lists

- There are many other kinds of linked lists applications.
- Consider a list of flights, each with a log of passengers.





Exercises

1. Write a function for one-way linked lists which swaps two elements.

   ```
   void swap(char * name1, char *name2, PP head);
   ```

   There is a starter file for this exercise in the `starters` directory.

This Page Intentionally Left Blank

Appendix A:

Software Tools

The cc Command

- Starting the C compiler can be done from:
  - the command line
  - an integrated environment.
- The C compiler command consists of three phases.
  - C preprocessor (CPP)
  - C compiler (CC)
  - C linker/loader (LD)

Different C Compilers

- There are many environments where C code is compiled.
- **UNIX** systems have a **cc** command for command line compiling.

$ cc program.c

- DOS, OS/2 and windows systems have command line compilers as well.
  - Borland (TCC) or (BCC)
  - Microsoft (CL)
  - Others
- Usually DOS, OS/2, and windows systems come with mouse or menu driven integrated environment packages consisting of:

  Compiler Editor Debugger Help

- We cannot describe them all so we give the fundamentals and the most frequently used options

Compiler Options

- By giving the proper option to the C compiler, you can produce either
  - .i Results of the preprocessor
  - .c Results of the compile
  - .exe (a.out) Results of the loader

- Command line examples
  - Just preprocess: producess .i

```
$ cc -P file.c
```

  - Preprocess and compile: prodices a **.o(bj)**

```
$ cc -c file.c
```

  - Produce an assembly listing .a

```
$ cc -A file.c
```

  - Pass a define to the program

```
$ cc -DSIZE=20 file.c
```

Compiler Options

- You can inform the compiler to stop whenever you wish: standard options include
  - Preprocessing
  - Compiling
  - Assembly listing
  - Defining constants
  - Producing code for a profiler

Conditional Compilation

- ifdef

```
#ifdef __STDC__
void *malloc(int);
#else
char *malloc(int);
#endif


#ifndef NAME
#define NAME
#endif
```

- For debugging

```
#ifdef DEVELOPING
printf("entering sub1: I = %d\n", i);
#endif
```

- Setting debugging levels

```
#if DEVEL > 1
printf("PRINT: I = %d\n", i);
#endif

#if DEVEL > 2
printf("PRINT: I = %d\n", i);
#endif

#if DEVEL > 1
printf("PRINT: I = %d\n", i);
#endif
```

Conditional Compilation

- C code can be conditionally compiled through the use of a few preprocessor directives
- Debugging can be enabled by placing sequences of #ifdefs at strategic places in your code
- Debugging levels can be established
  - `printfs` get executed depending upon level of debugging in effect

The `assert` Macro

- Using assertions

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

//  COPY argv[1] TO argv[2]

main(int argc, char **argv)
{
    FILE *fpin, *fpout;
    assert(argc == 3);
    fpin = fopen(argv[1],"r");
    assert(fpin != NULL);
    fpout = fopen(argv[2],"w");
    assert(fpout != NULL);
    while(( c = getc(fpin)) != EOF)
        putc(c, fpout);
}
```

- If an assertion is false, the following is displayed on the `stderr`.

```
Assertion failed:, argc == 3: file raise.c line 6
Abnormal program termination
```

The `assert` Macro

- When errors are detected during program execution
  - Print an error message
  - If possible, keep the program running
- When errors are detected during development
  - Print the error message
  - Exit
- The **assert macro** defined in `assert.h` can be used to detect conditions which you deem as errors. if so:
  - The program prints an error message
  - The program then exits

Libraries

- A library is a file maintained by a system utility
  - Consists of compiled files
  - Same subject matter
- Most C compilers come with several libraries
  - Portable I/O library
  - Math library
  - Graphics library
  - Others

Libraries

- A library on a computer system is a file. which has the following format.
  - A table of contents (or index)
  - A set of modules (or members)

- Systems differ as to exact components of a library file.

- Libraries are usually made up of modules from the same subject matter area.

Math library	Graphics library
String library	Windows library

- Some libraries are delivered with your C compiler.
  - On **Unix** systems, they exist in the directories

    `/lib` **`/usr/lib`**

  - In **DOS Windows**, and **OS/2**, they can be found under the main directory for your compiler.

    `/tc/lib` **`/msvc/lib`**

Libraries

- To build a library, construct the source file.

```
//  square.c

double square(double a)
{
        return(a * a);

}

$ cc -c square.c          // COMPILE IT

$ ar r mathlib.a square.o  // PLACE IN LIBRARY
```

- Repeat the process for the other functions.

```
//  sum_of_squares.c

double sum_of_squares(double a, double b)
{
    return(square(a) + square(b));
}


//  dist.c

#include <math.h>
double dist(int x1, int x2, int y1, int y2)
{
    double x = x1 # y1;
    double y = x2 # y2;
    return(sqrt(sum_of_squares(x, y)));
}
```

Libraries

- We will use the unix ar command to illustrate basic principles about libraries.
  - Saves compilation time
  - Easy to reuse functionality
- Suppose we want to build a library of math functions.
  - Build the first function and compile it.

```
$ cc -c square.c
```

  - Place it in a library.

```
$ ar r mathlib.a square.o
```

  - Repeat the process for other functions.

```
$ cc -c sum_of_squares.c
$ ar mathlib.a sum_of_squares.o
$ cc -c dist.c
$ ar mathlib.a dist.o
```

Header File Support

- Applications programs need to include the support header file to gain access to the prototypes.

```
//  mathlib.h
double dist(int, int, int, int);
double sum_of_squares(double, double);
double square(double);
```

- The program below uses the functionality from `mathlib.a`.

```
// distance.c
#include "mathlib.h"
main( )
{
    printf("%f\n", dist(0,0,5,5));
    printf("%f\n", dist(0,0,3,4));

}
```

- Note that failure to include the header file will leave strange results

```
//  A COMMON C BUG IS THE FAILURE TO
//  MAKE VISIBLE THE PROTOTYPE FOR atof
//
main( )
{
    char *s = "12345.65";
    printf("%f\n", atof(s));
}
```

Libraries

- Create a header file with the function prototypes for the functions placed in the library.
- When a program needs some of the functionality built into the library, name the library on the command line:

```
$ cc distance.c mathlib.a
```

  - Failure to name the library on the command line yields undefined name errors.
- The last phase of the **compiler** command searches libraries to find functions, which are referenced, but not supplied.
- The search for libraries consists of looking in:
  - standard places
  - libraries that you explicitly name on the command line

The `make` Command

- The procedure below is not smart.

```
$ cc prog.c fun1.c fun2.c fun3.c
fun2.c:
    errors

$ edit fun2.c
//
//  EVERYTHING IS NEEDLESSLY RECOMPILED
//
$ cc prog.c fun1.c fun2.c fun3.c
$
```

- The procedure below is smart.

```
$ cc prog.c fun1.c fun2.c fun3.c
fun2.c:
    errors

$ edit fun2.c
//
// NOTICE THAT fun2.c IS RECOMPILED
//
$ cc prog.o fun1.o fun2.c fun3.o
$
```

The `make` Command

- One executable usually is spread out over many source files.
- A change in one source need only effect the recompilation of that file.
- Other files can be linked to produce the executable.
- The Unix system was the first to automate the **smart** process above.
  - The program to automate program generation is called make.
  - Currently, most software development systems contain a version of make.
- We will explain the details of the make command

An Example Makefile

- An example **makefile** to produce the prog executable:

```
prog:    prog.o fun1.o fun2.o fun3.o
         cc prog.o fun1.o fun2.o fun3.o -o prog

prog.o:  proc.c header.h
         cc -c prog.c

fun1.o:  fun1.c
         cc -c fun1.c

fun2.o   fun2.c
         cc -c fun2.c

fun3.o   fun3.c header.h
         cc -c fun3.c
```

The make Command

- The make utility needs two pieces of information.
  - The dependency relationship between source files of a system
  - The most recent modification dates of those files
- make reads a description file (prepared by the user) in order to build a tree of dependencies existing between header, source, object, and executable files.
- **makefiles** consist of two types of lines.
  - Dependency lines

    `target: list of dependent files`

  - Rules lines

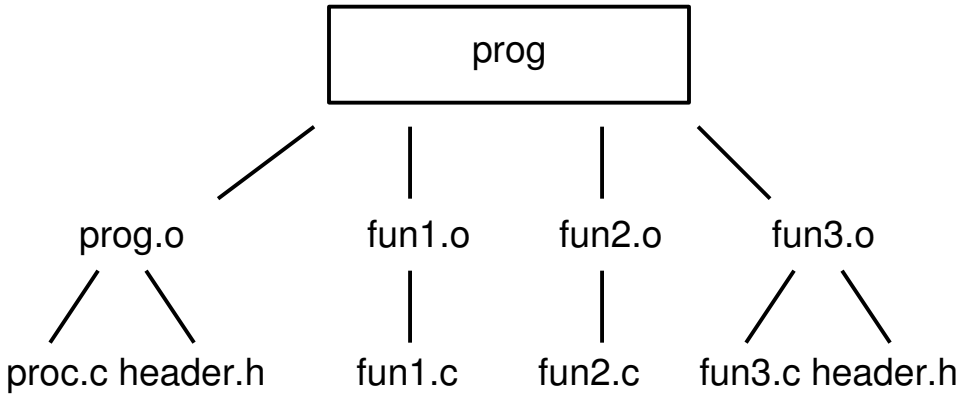
    `<-TAB-> SHELL COMMAND`

- Make can be executed in several ways.

```
$ make                    # DEFAULT IS makefile

$ make -f makefilename    # name your own makefile
```

The make Dependency Tree

The make utility builds a tree of dependencies.
The tree for the system above:



The make Command

- make reads the first target in the **makefile**.
  - For each **dependent** file on the target line, **make** searches for a line with this **dependent** as a **target**.
  - For example, **make** reads **prog** and finds that **prog** depends upon **prog.o**, which depends upon **prog.c**.
- When the search for targets expire, make checks the modification times for these files.
  - If a dependent file is **more recent** than a target all the **rules** for this dependency are executed.
  - **cc -c prog.c** would be executed if **prog.c was more recent than prog.o.**
  - This is turn would make **prog.o** more recent than **prog**.
- make has additional special features and built in rules.
- make can also be used to **make** documents.

Source Code Control System

- To place a file under the SCCS

```
$ admin -i software.c s.software.c
$ rm software.c
```

```
-i                initialize

s.software.c      THE SCCS FILE
```

- The process above creates an s. file, which consists of:
  - The original file in addition to
  - Some control information

s.software.c

CTRL	software.c 1.0	CTRL

- Use the get -e command to get a copy of the file for edit.

```
$ get -e s.software.c
```

  - This leaves a copy of the current version of `software.c` for you to edit!

Source Code Control System

- Software developers are constantly updating and releasing different versions of software.
- Some systems automate control over many versions of files by using file name and version number sequences.
- The Unix source code control system (SCCS) is a method of automatic revision control.
  - Keep the base file.
  - Keep each set of changes.
- The SCCS is a set of user level commands.
- When software is first released, place it under automatic revision control.
  - Under SCCS, use the admin command is used to start the revision control process.

After a Revision Cycle

- Use the `delta` command to enter the changes into the `s.software.c` file.

```
$ delta s.software.c
comments? (end with newline or ctrl -d)
$
```

- After a revision, the `s.software.c` file looks like:

s.software.c

CTRL	software.c 1.0	CTRL	DELTA 1.1	CTRL

  - Note: only the changes are saved.

- After a few revisions:

s.software.c

CTRL	software.c 1.0	CTRL	DELTA 1.1	CTRL	DELTA 1.2	CTRL

Source Code Control System

- Once the file is "gotten," you can edit (revise) the file you cannot modify the **s.** file.
- After modifying the `software.c` file, use the delta command to update the **s.** file.
- After the process is repeated several times, the s. file has many control sections so that any version can be reconstructed from the base version.
- Any version may be obtained by:

```
.$ get -e -r1.5 s.software.c
```

- Revision summaries may be obtained with:

```
$ prs s.software.c
```

This Page Intentionally Left Blank

Appendix B:

Library Functions

Building Command Strings

```
char command[100], line[100];

#ifdef DOS
strcpy(command, "copy ");
#else
strcpy(command, "cp ");
#endif

printf("name the input file ");
gets(line);
strcat(command,line);
strcat(command," ");
printf("name the output file ");
gets(line);
strcat(command,line);
system(command);
```

system

- The system function allows you to execute from your C program any command executable from the command line.

```
int system(char *);

system returns -1 on error and 0 otherwise

system("cls");

system("dir");
```

- Often, a command to be executed is built by using `strcpy` and `strcat`.

`exit` and `atexit`

- `exit` terminates the program and sends a value back to its parent.

```
void exit(int value);
```

  - Used mostly after error checking

- `atexit` provides a way to schedule events upon termination.

```
void fun(void);
void fun1(void);

main()
{
    atexit(fun);
    atexit(fun2);
    exit(0);
}

void fun()
{
    printf("1\n");
}

void fun2()
{
    printf("2\n");
}
```

`exit` and `atexit`

- The **`exit`** function is used to terminate your program.
  - Send a value to the parent of the exiting program
- Programs may be executed in a batch or script file.
- Programs may be started by another program.
- The `atexit` function allows you to execute any number of functions when your program terminates.
- During the running of the program, you must register functions using the `atexit` call.
  - Registered functions are stacked.
  - When the program terminates, the registered functions are executed.

signal

- The prototype for **signal** is in signal.h.

```
void signal(int sig_num, void (*response)(int));
```

- sig_num is the signal number as found in **signal.h**.

```
#define SIGFPE    8  /* Floating point trap     */
#define SIGILL    4  /* Illegal instruction     */
#define SIGINT    2  /* User interrupt          */
#define SIGSEGV  11  /* Mem access violation    */
#define SIGTERM  15  /* Software Termination    */
```

- response is the action to be taken upon receipt of this signal.

```
#define SIG_DFL  0   /*  Default action   */
#define SIG_IGN  1   /*  Ignore action    */
#define SIG_ERR #1   /*  Error return     */
```

  - The second argument is a pointer to a void function (taking one int arg).

- Some uses of signal

```
signal(SIGINT, SIG_IGN)        // IGNORE INTERRUPT

signal(SIGINT, fun);       // EXECUTE fun
```

signal

- A signal is a message (a small integer) sent to an executable program by the operating system.
- Signals can be initiated by:
  - a user (CTRL-C);
  - a program (ALARM); or
  - the operating system (KILL).
- When a program receives a signal, it can respond by:
  - ignoring the signal;
  - performing a system action (program termination); or
  - executing your own signal handler.
- The signal function is used to set a program response to a particular signal.
- There is no uniform support of signals among all operating systems.
- When your program begins, the default response (which is program termination) is established for all signals
- At various places in your program, executing the signal function sets a response to a particular signal.

strtok

- `strtok` tokenizes a string.
  - You specify the delimeters between the tokens.

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

main()
{
        char *s = "this is a string";
        char *x, *delim = " ";

        x = strtok(s,delim);

        while(x != NULL) {
            printf("%s\n",x);
            x = strtok(NULL,delim);
        }
}
```

strtok

- The `strtok` function tokenizes a string with respect to a set of delimeters that you supply.
  - Useful for breaking a string into words
  - Prototype for `strtok` from `string.h`

```
char *strtok(char *string, char *delimeters);
```

- `strtok` has an unusual interface.
- For any string which you wish to tokenize:
  - use `strtok(string, delims)` for first call; and
  - use `strtok(NULL, delim)` for subsequent calls.

memcpy and memset

● memcpy

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

int main(void)
{
    char src[] = "*****";
    char dest[] = "abcdefghijlkmnopqrstuvwxyz";
    char *ptr;

    printf("before memcpy: %s\n", dest);
    ptr = memcpy(dest, src, strlen(src));
    if (ptr)
        printf("after memcpy: %s\n", dest);
    else
        printf("memcpy failed\n");
    return 0;
}
```

● memset

```
#define NAMESIZE 20
struct example {
        char name[NAMESIZE];
        int age;
        char dept;
};

main()
{
    struct example item;
    memset(&item, 0, sizeof(struct example));
```

memcpy and memset

- The `memcpy` function copies **n** bytes from source to **dest**.

```
void *memcpy(void *dest, void *source, int n);
```

- `memset` copies **n** occurrences of **c** to **dest**.

```
void *memset(void *dest, int c; int n);
```

  - memset could be used to initialize a structure.

qsort

- A few invocations of the `qsort` function

```
#include <stdlib.h>

int strcmp(char *, char *);
int numcmp(int *, int *);
int dump(double *, double *);
int qcomp(char **, char **);

int x[10] = { 0,3,4,2,3,5,6,8,7,1};

main(int argc, char **argv)
{
     int i;
     //   AN ARRAY OF INTEGERS
     qsort(x,10,sizeof(int),numcmp);

     ++argv;
     ##argc;
     //
     // SORT STRINGS FORM THE COMMAND LINE
     //
     qsort(argv,argc,sizeof(char *),qcomp);
     for(i = 0; i < argc ; i++)
          printf("%s\n",argv[i]);
}

int numcmp(int *a, int *b)
{
     return(*a # *b);
}

int dump(double *a, double *b)
{
     return(*a # *b);
}

int qcomp(char **a, char **b)
{
     return(strcmp(*a,*b));
}
```

qsort

- The `qsort` function performs a quicksort on the data you supply.
- You must give `qsort` four arguments.
  - arg1 Address of the data to be sorted
  - arg2 How many items to be sorted
  - arg3 The sizeof each item
  - arg4 A comparison function

bsearch

```
#include <stdlib.h>
int numcmp(int *, int *);

int x[10] = { 0,3,4,2,3,5,6,8,7,1};

main(int argc, char **argv)
{
    int i, val, *ptr
    //
    //  SORT TO PUT THEM IN ORDER
    //
    qsort(x, 10, sizeof(int), numcmp);

    for ( i = 1; i < argc; i++) {
        val = atoi(argv[i]);
        ptr = bsearch(&val, x, 5, sizeof(int),
                    numcmp);
        if(ptr)
            printf("%d found at %d\n", val,
                    ptr # x);
    }
}
```

bsearch

- The `bsearch` function performs a binary search.
- Elements must be in ascending order.
- The `bsearch` function requires five arguments.
  - arg1 Address of the key to be found
  - arg2 Address of the table to be searched
  - arg3 Number of elements in the table
  - arg4 sizeof each element
  - arg5 Comparison routine

```
ptr = bsearch(&val, x, 5, sizeof(int), numcmp);
```

- `bsearch` returns a void * type.

strstr

- A simple version of a `grep` program

```
#include <string.h>
#include <stdio.h>
#define MAXLINE 100

main(int argc, char **argv)
{
    FILE *fp;
    char line[MAXLINE];
    int i;
    if (argc < 3) {
        printf("usage: grep pattern files..\n");
        exit(1);
    }
    for (i = 2; i < argc; i++)
        if(( fp = fopen(argv[i],"r")) == NULL)
            fprintf(stdout,"can't open %s\n",argv[i]);
        else
            while(fgets(line,MAXLINE,fp) != NULL)
                if(strstr(line, argv[1]) != NULL)
                    printf("%s",line);
}
```

strstr

- The `strstr` function takes two strings and determines whether the **second** is contained in the **first**.
  - If it isn't, **null** is returned.
  - Else, a pointer at the containing string is returned.

```
char *strstr(char *string, char *substring);
```

- Notice the use of `strstr` in `grep` utility.
  - All lines from argument files which match the pattern given as **argv[1]** are displayed.

strchr and strrchr

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

main(int argc, char **argv, char **envp)
{
    char *pc;

    if ( argc != 2) {
        printf("Usage: env argument\n");
        exit(1);
    }

    while(*envp != NULL) {
        if(strstr(*envp, argv[1])) {
            pc = strchr(*envp,'=');
            if(strlen(argv[1]) == (pc # *envp)){
                printf("%s\n", *envp);
                exit(1);
            }
        }
        ++envp;
    }
    printf("%s NOT FOUND\n", argv[1]);
}
```

strchr and strrchr

- `strchr` (`strrchr`) finds the **first (last)** occurrence of a character within a string.
- Each function returns a pointer to the character, or **null**, if there are no occurrences.

```
char *strchr(char *string, int character);
char *strrchr(char *string, int character);
```

- We use strchr to print the value of an environment variable whose name appears on the command line.

```
$ printenv PATH
/bin/usr:/bin:

$ printenv ME
ME: not in the environment

$ set ME=Michael

$ printenv ME
michael
```

- Environment strings are of the form:
  - STRING=VALUE

Data Validation Example

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 100

main()
{
     char line[MAXLINE], digits[11];
      int n,d;

     strcpy(digits,"0123456789");
     while(1) {
          printf("enter string ");
          gets(line);
          if(strcmp(line,"quit") == 0)
               break;
          else if((d=strspn(line, digits))==
strlen(line))
               printf("%s is all digits\n", line);
          else if((n=strcspn(line, digits))==
strlen(line))
               printf("%s has no digits\n", line);
          else {
               printf("%s is mixed\n", line);
               printf("first non digit is %d\n", d + 1);
               printf("first digit is %d\n", n + 1);
           }
      }
}
```

`strspn` and `strcspn`

- A common programming problem is data validation.
  - Is a string all of the same character class?
  - Does a string contain any of a class?
- These two problems are handled by the functions.

```
int strspn(char *string, char *class);
int strcspn(char *string, char *complement);
```

- `strspn` (`strcspn`) returns the length of the prefix of string that contains only characters in (not in) class.

Exercises

1. Write a function which determines if a string is of the exact form.

   .....ddddd.dd

   - (i.e) Any number of leading decimal digits followed by one decimal point followed by exactly two decimal digits.

2. Write a program that takes its own source and performs a frequency count on the words. The program should list the words alphabetically.

Appendix C:

File Access

I/O From Applications Programs

- Applications programs make requests of the operating system to:
  - open files
  - close files
  - read from files
  - write to files
- The OS:
  - manages the data structures
  - shuttles the data to and from the disk

System Calls vs. Library Calls

- The command line processor has aided us with file I/O.

```
C> program < input > output
```

- What if we needed to process many input or output files?

```
C> display file1 file2 file3
C> print thisfile thatfile otherfile
```

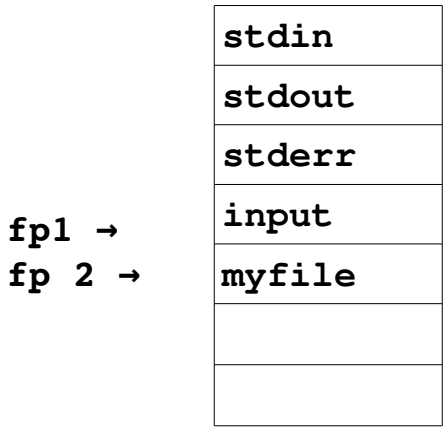
- There are generally two choices for I/O statements.
  - Host operating system (OS) calls
    - Bind your program to the host OS
    - Non portable
  - Standard C library calls
    - Portable interface
- I/O is accomplished by using a set of calls, which act as an agent between your program and the OS.

The `fopen` Function

- The `fopen` function is given a filename and a mode. It returns a file pointer to your application.

```
FILE *fp1, *fp2;
fp1 = fopen("input","r");
fp2 = fopen("myfile","w");
```

FILE TABLE



The `fopen` Function

- The OS limits the number of concurrently opened files.
  - A program can process any number of files.
  - A file must be opened before it can be processed.
- `fopen` requests the os to open the named file.
  - The prototype for `fopen` is:

    ```
    FILE *fopen(char *filename, char *mode);
    ```

    `fopen` returns:

    `FILE*` upon success / `NULL` upon failure
- The type `FILE*` is a **typedef** in `stdio.h`.
  - It is a pointer to an array of structures.
  - Each element of the array (the file table) holds information about an opened file.
  - You must provide a `FILE*` variable to receive the returned value from `fopen`.
  - This `FILE*` variable is then presented to all I/O statements.

Access Modes

- Some of the access modes are:

MODE	OPEN FILE FOR:	COMMENTS
"r"	read	error if not there
"w"	create	truncate if already exists
"a"	append	write at end of file
"r+"	read/write	error if does not exist
"w+"	read/write	truncated if exists
"a+"	read/write	only at end of file

- Open examples

```
FILE *fp;                  // fp: POINTER TO A FILE
char fname[256]
char mode[20];

fp = fopen("junk","r");   // junk BURIED INTO PROGRAM

fp = fopen(argv[1],"w");  // FILE FROM COMMAND LINE

//
// watch out for \\ to get a \ for DOS and OS/2 files
//

fp = fopen("c:\\c\\cnotes\\chap22","r");  // DRIVE
fp = fopen("/usr/include/stdio.h","r");   // OR PATH
```

Access Modes

- An access mode specifies the type of I/O to be performed on a file.
  - There are many combinations of access to a file.
- I/O is considered to be textual in mode.
  - If **binary** mode is desired, the character **b** must be part of the mode argument.
  - This style is used mostly with structures (not necessary on unix).
- The information, required by `fopen`, can be obtained from the user with the following sequence.

```
printf("enter a filename ");  // REQUEST INTERACTIVELY
gets(fname);                  // USER SUPPLIES NAME
printf("enter the mode ");
gets(mode);                   // AND PERHAPS MODE
fp = fopen(fname, mode);
```

Errors in Opening Files

- The `fopen` can fail for many reasons.
  - No permission for this type of access
  - The program wide file table is full
  - The file is not there (open for "r")
  - Usually results from spelling error

```
fp = fopen("myfil", "r"); // OOPS: myfile
```

- You should always check for error from `fopen`.

```
fp = fopen("myfile","r");

if( fp == NULL) {
    printf("couldn't open 'myfile'\n");
    exit(1);
}
```

OR

```
if ((fp = fopen("myfile", "r")) == NULL)
    printf("couldn't open 'myfile'\n");
    exit(1);
}
```

Errors in Opening Files

- The `fopen` function returns **null** upon failure.
- When you execute an `fopen`, **always** check for error.
- You can perform allowable I/O operations on a file:
  - after you have opened the file; or
  - after you have checked for errors.
- I/O is performed by a group of function calls (requests).
  - Each I/O function requires a **file pointer** (the one returned from `fopen`) as an argument.
  - This is how the function call knows upon which file the I/O is being performed.

Example: Copying a File

```
#include <stdio.h>

main(int argc, char **argv)
{
FILE *fpin, *fpout;
int c;

    // ERROR CHECKING
// 1. Make sure argument count is correct
if ( argc != 3) {
        fprintf(stderr, "usage: mycopy in out\n");
        exit(1);
    }

    fpin = fopen(argv[1],"r");

    // 2. Make sure input file is opened
if (fpin == NULL) {
        fprintf(stderr,"can't open %s\n",argv[1]);
        exit(2);
    }

    fpout = fopen(argv[2],"w");

    // 3. Make sure output file is opened
if (fpout == NULL) {
        fprintf(stderr,"can't open %s\n",argv[2]);
        exit(3);
    }
    // END OF ERROR CHECKING

while(( c = getc(fpin)) != EOF)
        putc(c,fpout);

exit(0);
}
```

I/O Library Calls

- Input functions

```
c = getc(fp);                  // similar to getchar()
fgets(line,MAX,fp);            // similar to gets(line)
fscanf(fp,"formats",items);    // similar to scanf
```

- Output functions

```
putc(c, fp);                   // similar to putchar(c)
fputs(line,fp);                // similar to puts(line)
fprintf(fp,"formats", items); // similar to printf
```

  - Each function above performs I/O on the file associated with the **file pointer (fp).**
  - Each has a direct analog with a function dedicated to either the standard input or standard output.

- `stdio.h` defines three constants of type `FILE *`

`stdin`	used with any input function
`stdout`	used with any output function
`stderr`	used with any output function

  - These names can be used as file pointer constants. In this way, the functions above can also read from the standard files.

```
FILE *fp = stdin;
if ( argc == 2 )
    fp = fopen(argv[1],"r");
else{
    printf("usage: prog filename\n");
    exit(1);
}
```

Character Input vs. Line Input

- Character at a time

```
while(( c = getc(fpin)) != EOF)
    putc(c,fpout);
```

- Line at a time

```
define MAX 100
char line[MAX];

while(fgets(line, MAX, fpin)) != NULL)
    fputs(line,fpout);
```

Character Input vs. Line Input

- The last example performed I/O a character at a time.
- We could have processed the data a line at a time.
  - This would have required a few changes.
- `fgets` is somewhat like `gets` but more capable.
  - Will not overwrite the array
  - Can read from any file (not just stdin)
  - Retains the newline character
  - Returns **NULL** at end of file
- In general, it makes little difference between reading by character or by line.
- In some problems, the amount of data accessed by each input statement is obvious from the problem itself.
  - Sorting lines
  - Reading records
  - Counting character types

Interpreting Input

- Interpreting a line of input as a number
  - Get the line:

```
char line[100];

printf("enter a number ");
gets(line);
```

  - Convert it to a number using:

```
int atoi(char *);          // ascii to int
long atol(char *);        // ascii to long
double atof(char *);      // ascii to float
```

- Reading several values on the same line is more difficult.
  - Get the line.
  - Parse the line into the various fields.

The `scanf` Function

- The I/O thus far has been character oriented.
  - A line of characters
  - A character
- Suppose we need to interpret a line as a number.
  - Get the line using `gets`
  - Convert it using a function
- It is harder to interpret two values on the same line.

```
printf("enter your age and weight ");
gets(line);
```

A name and a number

```
printf("enter your name and age ")
```

Other combinations

- These problems could be solved by asking the user for one piece of input per line.
  - This might be unnatural.
  - It is also awkward when there are many fields.

The `scanf` Function

- Input a name and a number

```
char name[MAX];
int number;
scanf("%s %d", name, &number);
```

- Input an interest rate and a starting amount

```
double rate;
double value;
scanf("%lf %lf", &rate, &value);
```

- Common error: forgetting address symbol

```
int number;
char name[NAMESIZE];
printf("enter a name and a number ");
n = scanf("%s %d", name, number); // FORGOT &
if( n != 2)
    fprintf(stderr,"bad data\n");
```

- Any of the input lines below will cause `scanf` to return the value 1

```
mike jane
mike 253ab
mike ab352
```

- The next input statement would continue from the offending character.
- The line below may also cause problems

```
mike  25  234
```

(Extra item is read by the next input statement)

The `scanf` Function

- `scanf` is the analog to `printf`.
  - Scans the input using blanks and tabs as data field separators
  - Requires a control string
  - Governs how much data is to be read
  - Specifies the type of the data
- It returns the number of correctly matched **(data to format)** items (or **EOF**).
  - Difficult to do error checking
- A common `scanf` error is to forget the address symbol.

`scanf` Variants

- `sscanf` is very useful for problems such as the one below.

```
int number;
char line[MAX];
char name[NAMESIZE];
char dum[DUMMYSIZE];    // GOBBLE EXTRA INPUT

printf("enter a name and a number ");
fgets(line, MAX, stdin);
n = sscanf(line, "%s %d %s", name, &number, dum);
if( n != 2)
    fprintf(stderr,"incorrectly formatted data\n");
```

- First, read the entire line into line.
- Next, separate the output using `sscanf`.
- Finally, error check.

`scanf` Variants

- The list below summarizes the `scanf` variations.

```
char info[100];
char name[NAMESIZE];
int x = 20;
FILE *fp;


// DATA COMES FROM THE STANDARD INPUT

scanf("%s %d\n", name, &x);

// DATA COMES FROM THE File ASSOCIATED WITH fp

fscanf(fp, "%s %d\n", name, &x);

// DATA COMES FROM THE STRING info

sscanf(info, "%s %d", name, &x);
```

- `sscanf` is like `scanf` except that the input comes from a string.

scanf Variants

- fscanf example

```
FILE *fp;
char fname[20], lname[20];

fp = fopen("inputfile", "r");
if( fp == NULL) {
    error("fopen error\n");
n = fscanf(fp,"%s %s\n", fname, lname);
if( n != 2)
    error("scanf error\n");
```

- scanf example

```
#include <stdio.h>
#define WORDSIZE 40

main()
{
    char word[WORDSIZE];

    while(scanf("%s", wd) != EOF) {

//   PROCESS THE WORD, e.g
//  1)   PRINT IT
//  2)   REMOVE PUNCTUATION
//  3)   etc

printf("%s\n",wd);
    }
}
```

`scanf` Variants

- `fscanf` is like `scanf` except the input comes from the file associated with the file pointer.
- While `scanf` does not provide reliable error checking, it is safe to use if you need to input string information.
- A nice use of `scanf` is to read a file a word at a time.

`printf` Variants

- `sprintf` is useful for "gluing" unlike data together.

```
char string[100];
char temp1[10];
char temp2[10];
int number;

strcpy(temp1,"hello");
strcpy(temp2,"goodbye");
number = 356;
sprintf(string,"%s%d%s", temp1, number, temp2);
n = strlen(string);    // n = 15
```

- `fprintf` is useful for sending data to error files.

```
fprintf(stderr,"error message #1\n");
```

printf Variants

- printf has analogous variants to scanf.
  - printf sends data to the display.
- sprintf data to a string
  - useful for concatenating unlike data

```
int x;
char name[NAMSESIZE], info[SIZE];
sprintf(info,"data %d\n", x);
```

- fprintf sends data to a file associated with a file pointer.

```
char info[100];
char name[NAMESIZE];
int x = 20;
FILE *fp;

fp = fopen("file","r+");  // open for read/write

//  SEND DATA TO file
fprintf(fp,"data %d\n", x);

//  SEND DATA TO THE STANDARD ERROR
fprintf(stderr,"error on file1\n");
```

The `fclose` Function

```
#include <stdio.h>

main(int argc, char **argv)
{
FILE *fp;
int c, ct = 0;

while ( ##argc > 0 ) {
        fp = fopen(*++argv,"r");  // OPEN NEXT FILE
        if ( fp == NULL){
fprintf(stderr,"can't open %s\n",
       *argv);
continue;
        }

        ct = 0;
        while (( c = getc(fp)) != EOF)
if (c == '\n')
ct++;

        fclose(fp);// CLOSE FILE
        printf("%d %s\n",ct, *argv);
}
}
```

The `fclose` Function

- There is a limit to the number of concurrent open files.
  - `fopen` will fail if you exceed this limit.
- To the operating system, opening a file means occupying a slot in a data structure.
  - This is why you should always close the file when you are done processing it (i.e to free the slot).
- The function `fclose` is provided for this.

```
int fclose(FILE *);
```

- Some systems provide a function named `fcloseall`.

```
void fcloseall(void);
```

- It is common to write code, which loops through a set of files performing some operation on each file.
  - For these programs, it is good programming practice to close the file when you are through processing it.

Servicing Errors

- Some I/O functions return `eof` both on error and end of file.
- The functions `feof` and `ferror` can be used to distinguish these two conditions.
  - `feof` returns **true** at **end-of-file**/**false** otherwise
  - `ferror` returns **true** on **error**/**false** otherwise

```
while(! foef(fp))                    // NOT EOF
if (( c = getc(fp)) == EOF)
// MUST BE ERROR
```

- `fopen` returns **null** upon failure. if you need to know the reason for the failure, you need to dig more
  - The OS maintains a variable called `errno`.
  - `errno` obtains a value when a system call fails even if the system call is invoked from a library call.
- You can access `errno` by using the **extern** declaration.

```
extern int errno;
```

- You can also another system variable, which has access to system wide error messages.
  - `extern char *sys_errlist[];` is an array of character pointers each points to an error message. you can index this array with `errno`.

Servicing Errors

- Your error processing may look like this.

```
fp = fopen(filename,mode);
if(fp == NULL) {
    if(errno == ENOENT)
do one thing
    else
do another
}
```

- An excerpted copy of `errno.h`

```
#define ENOENT  2  /* No such file or directory    */
#define EMFILE  4  /* Too many open files  */
#define EACCES  5  /* Permission denied*/
#define EDOM    33 /* Math argument*/
#define ERANGE  34 /* Result too large*/
```

Servicing Errors

- Exact system error messages and their codes can be found in the file `errno.h` in the **include** directory.
- **On Unix** systems, this directory is always

  `/usr/include`.
- On **DOS** systems, the exact directory will depend upon the installation procedure for the compiler.
  - Check the value of the **include** environment variable.

Application for Binary I/O

- The application begins by displaying the following.

```
 SELECT THE NUMBER OF YOUR CHOICE

0) QUIT THE PROGRAM
1) CREATE DATA BASE
2) PRINT DATA BASE
3) RETRIEVE

==>
```

- To demonstrate **binary I/O**, the following structure type will be used.

```
//
//employee.h
//

#define NAMESIZE 20

struct employee {
char name[NAMESIZE];
float pay;
char dept;
};

typedef struct employee WORKER, *PWKR;
```

Binary I/O

- There are two methods of writing data to a file.
  - Convert from in memory format to character (stream I/O)
    - Human readable
  - Copy exactly as it is in memory (binary I/O)
    - Faster
    - Machine readable
- Binary I/O is used most often with structures, but is not limited to structures.
- The functions will be demonstrated by developing a typical financial application dealing with structures

The `main` Function - Code

```
#include "employee.h"
main( )
{
char line[MAXLINE];
int selection;

while(1) {
        selection = menu();

        switch(selection) {
case QUIT:
exit(0);
case CREATE_DB:
create_db();
break;
case PRINT_DB:
print_db( );
break;
case RETRIEVE:
retrieve_db( );
break;
default:
printf("SHOULD BE IMPOSSIBLE\n");
break;
        }

        printf("RETURN to continue\n");
        fgets(line,MAXLINE,stdin);
    }
}
```

The `main` Function

- The `main` function a merely a switch.
  - Calls **menu** to get a user response
- The `menu` function prints the menu and returns the response.

create_db Function - fwrite

```
void create_db(void)
{
WORKER person;
FILE *fp;
     char fname[FILENAMESIZE], line[100];
     int number, i;

printf("enter filename ");
gets(fname);
     if((fp = fopen(fname,"wb")) == NULL) {
          printf("can't open %s\n", fname);
          return;
}

     printf("how many records to input? ");
     number = atoi(gets(line));
     for (i = 0; i < number; i++) {
          fill(&person);
          fwrite(&person, sizeof(WORKER),1,fp);
}
     fclose(fp);
}
```

- Other examples of `fwrite`

```
     int i, n[10];
     WORKER x, a[10];                    // WRITE:

     fwrite(&i,sizeof(int),1,fp);        // INT
     fwrite(n,sizeof(int),10, fp);       // AN ARRAY
     fwrite(n,sizeof(int),5,fp);         // FIRST HALF
     fwrite(&x,sizeof(WORKER),1,fp);     // WORKER
     fwrite(a,sizeof(WORKER),10,fp);     // ARRAY
     fwrite(a,sizeof(WORKER),5,fp);      // FIRST HALF
     fwrite(a + 5,sizeof(WORKER),5,fp);// LAST HALF
```

fwrite

- Records are typed by the user and then copied to disk using `fwrite`.

```
int fwrite(void *data, int size, int hmany, FILE
*file);
```

- `fwrite` requires four arguments.

```
arg1 = ADDRESS OF THE DATA TO BE WRITTEN

arg2 = SIZE OF ONE PIECE OF DATA

arg3 = HOW MANY PIECES

arg4 = FILE POINTER
```

- `fwrite` returns the number of objects written.

print_db Function - fread

- Reading one record at a time

```
void print_db(void)
{
WORKER person;
  FILE *fp;
  char fname[FILENAMESIZE];

  printf("enter filename ");
gets(fname);
if((fp = fopen(fname,"rb")) == NULL) {
        printf("can't open %s\n", fname);
        return;
}
  while((fread(&person, sizeof(WORKER),1,fp)) >0)
        output(&person);

  fclose(fp);
}
```

- Reading more than one record at a time

```
WORKER bank[CHUNK];
int num, i;

while((num=fread(bank,sizeof(WORKER),CHUNK,fp))>0)
for (i = 0; i < num; i++)
        process each record!!
```

fread

- `print_db` reads the file created by `create_db`.

```
int fread(void *data, int size, int hmany, FILE
*file);
```

- Same argument structure as `fwrite`
- `fread` returns the number of objects read.
  - **Not** an error, if different from `hmany`
  - **0** is returned at end of file
- `print_db` reads **one** record at a time
  - You can read as many as you want if you have arranged for a big enough buffer
- Note that both `create_db` and `print_db` are void functions
  - Premature exit from a void function should use `return`;

retrieve_db Function

```
void retrieve_db(void)
{
WORKER person;
     char fname[FILENAMESIZE], pname[NAMESIZE];

     printf("enter file name ");
     gets(fname);
     if (( fp = fopen(fname,"rb")) == NULL) {
          printf("can't open %s\n",fname);
          return;
     }
     while(1) {
          printf("which record? ('quit' to end) ");
          gets(pname);
          if (strcmp(pname,"quit") == 0)
break;

          while((fread(&person, sizeof(WORKER),1,fp))>0)
if(strcmp(pname,person.name) == 0)
output(&person);

          fseek(fp,0L,0);
    }
    fclose(fp);
}
```

- Other **fseek** examples

```
     long int n = 5;
     long int size = sizeof(WORKER);

     fseek(fp,0L,0);            // SEEK TO BEGINNING
     fseek(fp,0L,2);            // SEEK TO END
     fseek(fp,-size,1);         // BACK UP ONE RECORD
     fseek(fp, size * n, 0)     // SKIP FIRST 5 RECORDS
```

- Reading the last record of a file

```
     fp = fopen(    ,    );   // OPEN THE FILE
     if ( fp == NULL)
     error();
     fseek(fp, -size, 2);     // SEEK TO LAST RECORD
     fread(    ,    ,    ,);  // READ IT
```

fseek

- `retrieve_db` is similar to `print_db` except that only selected records are printed.
- Files are normally read and written sequentially.
  - The operating system keeps a pointer to the next record.
  - You can control this pointer with the `fseek` function

```
int fseek(FILE *file, long offset, int origin);
```

- `fseek` positions the file **offset** many bytes from **origin**.
  - **origin** can be given as a `#define` from `stdio.h`.

```
#define SEEK_SET  0  // FROM BEGINNING
#define SEEK_CUR  1  // FROM CURRENT POS
#define SEEK_END  2  // FROM END OF FILE
```

- With `fseek` every file in effect becomes a random file.
  - The next read or write is independent of the previous one.

`fflush` and `ftell`

- Updating a record in a file

```
//
//  OPEN THE FILE
//
long int size = sizeof(WORKER);

while((fread(&person, sizeof(WORKER),1,fp)) > 0)
    if(strcmp(pname,person.name) == 0)  {
// modify record
fseek(fp, -size, 1);
fwrite(&person,sizeof(WORKER,1,fp);
fflush(fp);
    }
```

- `ftell` tells the byte position of the file.

```
long int size, recs;

fp = fopen(file,mode);
fseek(fp,0L,2);
size = ftell(fp);
recs = size / sizeof(WORKER);
printf("file %s is %ld bytes\n", file, size);
printf("file %s has %ld records\n", file, recs);
```

`fflush` and `ftell`

- It is easy to add functionality to this application.
- One addition might be to modify a record.
  - Read one record at a time.
  - If this is the record to be modified
  - Modify in memory
  - Backup a record and write it back to disk
- The `fflush` causes output to immediately be flushed rather than having the output buffered
- Another addition might be to append to an existing file.
  - Open for append and copy from user to disk.
- Another might be to print the number of records in the file.
  - The `ftell` function gives the byte position of the file as a long integer.

```
long ftell(FILE *file)
```

Exercises

1. Write the program **compare** which displays information about the two files named on the command line.

```
C> compare file1 file2
file1 and file2 are the same
C> compare file1 file3
file1 and file3 are different
C>
```

2. Write the program **glue** which glues together files named on the command line and sends them to the display.

```
C> type file1
hello there
C> type file2
how are you
C> glue file1 file2
hello there
how are you
C>
```

3. Add a variation to (2): If the last named file on the command line is prefixed with a '**+**' the output should go there and not to the display.

```
C> glue file1 file2 +output
C> type output
hello there
how are you
C>
```

4. Write a program which exchanges the first and last record of a file.

5. Write a program which receives a filename and a number and prints the last **number** many records from the file.

```
C> print datafile 3     # print last 3 records
```

This Page Intentionally Left Blank