Problem Set 1

Writing, compiling, and debugging programs. Preprocessor macros. C file structure. Variables. Functions and program statements. Returning from functions.

Problem 1.1

(a) What do curly braces denote in C? Why does it make sense to use curly braces to surround the body of a function?

(b) Describe the difference between the literal values 7, "7", and '7'.

(c) Consider the statement

```
double ans = 10.0 + 2.0 / 3.0 − 2.0 ∗ 2.0;
```

Rewrite this statement, inserting parentheses to ensure that ans = 11.0 upon evaluation of this statement.

Solution 1.1

(a) Curly braces "{}" denote the scope in C. It also denotes a block of code. They are used to define the beginning and end of a code block, such as a function, loop, conditional statement, or any other compound statement.

(b) 7 is integer.
"7" is string literal.
'7' is a character.

(c) double ans = 10.0 + (2.0 / ((3 - 2.0) * 2.0));

Problem 1.2

Consider the statement

double ans = 18.0 / squared(2+1);

For each of the four versions of the function macro squared() below, write the corresponding value of ans.

1.#define squared(x) x*x
2.#define squared(x) (x*x)
3. #define squared(x) (x)*(x)
4. #define squared(x) ((x)*(x))

Solution 1.2

1. #define squared(x) x*x
ans = 18.0/2+1*2+1 = 12.00
2. #define squared(x) (x*x)
ans = 18.0/(2+1*2+1) = 3.60
3. #define squared(x) (x)*(x)
ans = 18.0/(2+1)*(2+1) = 18.00
4. #define squared(x) ((x)*(x))
ans = 18.0/((2+1)*(2+1)) = 2.0

Problem 1.3

Write the "Hello, 6.087 students" program described in lecture in your favorite text editor and compile and execute it. Turn in a printout or screen shot showing

- the command used to compile your program
- the command used to execute your program (using gdb)
- the output of your program

Solution 1.3

```
#include <stdio.h>
#include <conio.h>
int main(int argc, char* argv[]) {
        printf("Hello, %.3f students", 6.087);
        return 0;
}
```

• gcc -o hello hello.c
• .\hello
• Hello, 6.087 students

Problem 1.4

The following lines of code, when arranged in the proper sequence, output the simple message "All your base are belong to us."

1. return 0;
2. const char msg[] = MSG1;
3. }
4. #define MSG1 "All your base are belong to us!"
5. int main(void) {
6. #include <stdio.h>
7. puts(msg);

Write out the proper arrangement (line numbers are sufficient) of this code.

Solution 1.4

6, 4, 5, 2, 7, 1, 3

```
6. #include <stdio.h>
4. #define MSG1 "All your base are belong to us!"
5. int main(void) {
2.      const char msg[] = MSG1;
7.      puts(msg);
1.              return 0;
3. }
```

Problem 1.5

For each of the following statements, explain why it is not correct, and fix it.

(a) #include <stdio.h>;

(b)
```
int function(void arg1) {
        return arg1-1;
}
```

(c)
```
#define MESSAGE = "Happy new year!"
puts(MESSAGE);
```

Solution 1.5

(a) Semicolon is not allowed at the end of preprocessors.

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

(b) If a function has no argument as input then it could be empty or void.

```
int function(void) {
        return arg1-1;
}
```

(c) Wrong macros definition.

```
#define MESSAGE "Happy new year!"
puts(MESSAGE);
```

Problem Set 2

Types, operators, expressions

Problem 2.1

Determine the size, minimum and maximum value following data types. Please specify if your machine is 32 bit or 64 bits in the answer.

- char
- unsigned char
- short
- int
- unsigned int
- unsigned long
- float

Hint: Use sizeof() operator, limits.h and float.h header files

Solution 2.1

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

int main() {
    // char
    printf("Size of char: %lu\n", sizeof(char));
    printf("Minimum value of char: %d\n", CHAR_MIN);
    printf("Maximum value of char: %d\n", CHAR_MAX);

    // unsigned char
    printf("Size of unsigned char: %lu\n", sizeof(unsigned char));
    printf("Minimum value of unsigned char: 0\n");
    printf("Maximum value of unsigned char: %u\n", UCHAR_MAX);

    // short
    printf("Size of short: %lu\n", sizeof(short));
    printf("Minimum value of short: %d\n", SHRT_MIN);
    printf("Maximum value of short: %d\n", SHRT_MAX);

    // int
    printf("Size of int: %lu\n", sizeof(int));
    printf("Minimum value of int: %d\n", INT_MIN);
```

```
    printf("Maximum value of int: %d\n", INT_MAX);

    // unsigned int
    printf("Size of unsigned int: %lu\n", sizeof(unsigned int));
    printf("Minimum value of unsigned int: 0\n");
    printf("Maximum value of unsigned int: %u\n", UINT_MAX);

    // unsigned long
    printf("Size of unsigned long: %lu\n", sizeof(unsigned long));
    printf("Minimum value of unsigned long: 0\n");
    printf("Maximum value of unsigned long: %lu\n", ULONG_MAX);

    // float
    printf("Size of float: %lu\n", sizeof(float));
    printf("Minimum positive value of float: %e\n", FLT_MIN);
    printf("Maximum positive value of float: %e\n", FLT_MAX);

    return 0;
}
```

Problem 2.2

Write logical expressions that tests whether a given character variable c is

- lower case letter
- upper case letter
- digit
- white space (includes space,tab,new line)

Solution 2.2

```
#include <stdio.h>
#include <conio.h>
int main() {
    char letter;
    fflush(stdout);
    printf("Enter a input : ");
    scanf("%c", &letter);

    if (letter >= 'a' && letter <= 'z')
        printf("lower letter\n");
    else if (letter >= 'A' && letter <= 'Z')
        printf("upper letter\n");
    else if (letter >= '0' && letter <= '9')
        printf("digit\n");
    else if (letter == ' ' || letter == '\n' || letter == '\t')
        printf("white space\n");
    return 0;
}
```

Problem 2.3

Consider int val=0xCAFE; Write expressions using bitwise operators that do the following:

- test if at least three of last four bits (LSB) are on
- reverse the byte order (i.e., produce val=0xFECA)
- rotate fourbits (i.e., produce val=0xECAF)

Solution 2.3

```
#include <stdio.h>
int main() {
        // Hexadecimal Number
        int val = 0xCAFE;  // 1100 1010 1111 1110

        // Test if at least three of the last four bits (LSB) are ON(1)
        if ((val & 0x0F) >= 0x07) {
                printf("At least three of the last four bits are on\n");
        }

        // Reverse the byte order (produce val = 0xFECA)
        val = ((val & 0x00FF) << 8) | ((val & 0xFF00) >> 8); // FE00 | 00CA = FECA
        printf("Reversed byte order: 0x%X\n", val);

        // Rotate four bits (produce val = 0xECAF)
        val = ((val << 4) | (val >> 12)) & 0xFFFF; // C AFE0 | C = AFEC
        printf("Rotated four bits: 0x%X\n", val);

        return 0;
}
```

Problem 2.4

Using precedence rules, evaluate the following expressions and determine the value of the variables(without running the code).

Also rewrite them using parenthesis to make the order explicit.

- Assume (x=0xFF33,MASK=0xFF00).Expression: c=x & MASK == 0;
- Assume (x=10,y=2,z=2;).Expression: z=y=x++ + ++y∗2;
- Assume (x=10,y=4,z=1;).Expression: y>>= x&0x2 && z

Solution 2.4

- c = (x & MASK) == 0;
  c = (0xFF33 & 0xFF00) == 0;
  c = 0;
- z = y = (x++) + (++y * 2);
  z = y = (10) + (3 * 2);
  z = y = 10 + 6;
  z = y = 16;
- y >>= (x & 0x2) && z;
  y = 4 >> (10 & 0x2) && 1;
  y = 4 >> 1;
  y = 2;

Problem 2.5
Determine if the following statements have any errors. If so, highlight them and explain why.

- int 2nd value=10;
- Assume (x=0,y=0,alliszero=1). alliszero = (x=1) && (y=0);
- Assume (x=10,y=3,z=0;). y=++x+y;z=z−−>x;
- Assume that we want to test if the last four bits of x are on. (int MASK=0xF; ison=x&MASK==MASK)

Solution 2.5

- Variable names can not start with digit, space, and special character except _.
- int alliszero = (x=1 && (y=0);
- y = ++x + y;
  z = z-- > x;
- int MASK = 0xF;
  int ison = ((x & MASK) == MASK);

Problem Set 3

Control flow. Functions. Variable scope. Static and global variables.

I/O: **printf** and **scanf**. File I/O. Character arrays. Error handling. Labels and goto.

Problem 3.1

Code profiling and registers. In this problem, we will use some basic code profiling to examine the effects of explicitly declaring variables as registers. Consider the fibonacci sequence generating function fibonacci in **prob1.c.** Which is reproduced at the end of this problem set (and can be downloaded from Stellar). The **main()** function handles the code profiling, calling **fibonacci()** many times and measuring the average processor time.

(a) First, to get a baseline (without any explicitly declared registers), compile and run **prob1.c**. Code profiling is one of the rare cases where using a debugger like **gdb** is discouraged, because the debugger's overhead can impact the execution time. Also, we want to turn off compiler optimization. Please use the following commands to compile and run the program: dweller@dwellerpc:~$ *gcc -O0 -Wall prob1.c -o prob1.o*

dweller@dwellerpc:~$ *./prob1.o*

Avg. execution time: 0.000109 msec ← example output

dweller@dwellerpc:~$

How long does a single iteration take to execute (on average)?

Solution 3.1(a)

Prob1.c

```c
// prob1.c

#include <stdio.h>

#include <time.h>



void fibonacci(int n) {

    int a = 0, b = 1, c;

    printf("Fibonacci Series up to %d terms: \n", n);

    for (int i = 0; i < n; i++) {

        printf("%d, ", a);

        c = a + b;

        a = b;

        b = c;

    }

}



int main() {

    clock_t start, end;

    double cpu_time_used;

    start = clock(); // start time.



    int terms = 20;

    fibonacci(terms);
```

```
    end = clock();

    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;

    printf("\nTotal Time : %f\n", cpu_time_used);



    return 0;

}
```

PS D:\Program\C> gcc -O0 -Wall prob1.c -o prob1.exe

PS D:\Program\C> ./prob1.exe

Fibonacci Series up to 20 terms:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181,

Total Time : 0.002000

PS D:\Program\C>

(b) Now, modify the **fibonacci()** function by making the variables a, b, and c register variables. Recompile and run the code. How long does a single iteration take now, on average? Turn in a printout of your modified code (the **fibonacci()** function itself would suffice).

Solution 3.1(b)

Prob2.c (modified prob1.c)

```
// prob2.c

#include <stdio.h>

#include <time.h>



void fibonacci(int n) {

    register int a = 0, b = 1, c;

    printf("Fibonacci Series up to %d terms: \n", n);
```

```c
    for (int i = 0; i < n; i++) {
        printf("%d, ", a);
        c = a + b;
        a = b;
        b = c;
    }
}

int main() {
    clock_t start, end;
    double cpu_time_used;
    start = clock(); // start time.

    int terms = 20;
    fibonacci(terms);

    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("\nTotal Time : %f\n", cpu_time_used);

    return 0;
}
```

PS D:\Program\C> gcc -O0 -Wall prob2.c -o prob2.exe

PS D:\Program\C> ./prob2.exe

Fibonacci Series up to 20 terms:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181,

Total Time : 0.001000

PS D:\Program\C>

(c) Modify the **fibonacci()** function one more time by making the variable and also a register variable. Recompile and run the code once more. How long does a single iteration take with all four variables as register variables?

Solution 3.1(c)

Prob3.c (modified prob1.c)

```c
// prob3.c

#include <stdio.h>

#include <time.h>



void fibonacci(int n) {

    register int a = 0, b = 1;

    int c;

    printf("Fibonacci Series up to %d terms: \n", n);

    for (int i = 0; i < n; i++) {

        printf("%d, ", a);

        c = a + b;

        a = b;

        b = c;
```

```
    }
}


int main() {
    clock_t start, end;
    double cpu_time_used;
    start = clock(); // start time.


    int terms = 20;
    fibonacci(terms);


    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("\nTotal Time : %f\n", cpu_time_used);


    return 0;
}
```

PS D:\Program\C> gcc -O0 -Wall prob3.c -o prob3.exe

PS D:\Program\C> ./prob3.exe

Fibonacci Series up to 20 terms:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181,

Total Time : 0.002000

PS D:\Program\C>

(d) Comment on your observed results. What can you conclude about using registers in your Code?

Solution 3.1(d):

A normal variable is stored in the computer's memory and can be accessed by its memory address. These variables are used to store data during the execution of a program and are accessed using their variable names.

A register variable is a type of variable that is stored in the CPU registers instead of the computer's memory. CPU registers are high-speed storage locations within the processor that can be directly accessed by the CPU. By declaring a variable as a register variable, you are requesting the compiler to store that variable in a CPU register, which can result in faster access to the variable's value.

The use of register variables is a way to optimize the performance of certain parts of a program by reducing memory access time. However, it's important to note that the decision to store a variable in a register is ultimately up to the compiler, and the use of the register keyword is more of a hint to the compiler rather than a strict command.

In summary, the main difference between a normal variable and a register variable is their storage location: normal variables are stored in memory, while register variables are stored in CPU registers for faster access.

Problem 3.2

We are writing a simple searchable dictionary using modular programming. First, the program reads a file containing words and their definitions into an easily searchable data structure. Then, the user can type a word, and the program will search the dictionary, and assuming the word is found, outputs the definition. The program proceeds until the user chooses to quit. We split the code into several files: **main.c, dict.c**, and **dict.h**. The contents of these files are described briefly below.

Answer the following questions based on the above program structure.

(a) In implementing this program, you want to access the global variable the dictionary from **main.c**, as well as from **dict.c**. However, due to the header file's inclusion in both source documents, the variable gets declared in both places, creating an ambiguity. How would you resolve this ambiguity?

(b) Now, suppose you want to restrict the dictionary data structure to be accessible only from functions in **dict.c**. You remove the declaration from **dict.h**. Is it still possible to directly access or modify the variable from **main.c**, even without the declaration in **dict.h**? If so, how would you ensure the data structure variable remains private?

(c) Congratulations! You're done and ready to compile your code. Write the command line that you should use to compile this code (using **gcc**). Let's call the desired output program **dictionary.o**.

Solution 3.2

(a) By using extern in the header file to declare the global variable and defining it in one source file, you avoid multiple definitions and resolve the ambiguity when including the header file in different source files.

(b) By removing the declaration of the dictionary from dict.h and making it static within dict.c, you prevent direct access from main.c while maintaining the data structure variable's privacy, allowing access and modification only through functions within dict.c.

(c) gcc -c dict.c -o dictionary.o

Problem 3.3

Both the for loop and the do-while loop can be transformed into a simple while loop. For each of the following examples, write equivalent code using a while loop instead.

```
(a) int factorial ( int n) {
int i , ret = 1 ;
for ( i = 2 ; i <= n ; i++)
ret *= i ;
return ret ;
}
```

```
(b) #include <stdlib.h>
double rand double () {
/* generate random number in [ 0 , 1 ) */
double ret = (double ) rand ();
return ret /(RANDMAX+1);
}
int samplegeometric rv (double p) {
double q ;
int n = 0 ;
do {
q = rand double ();
n++;
} while ( q >= p );
return n ;
}
```

Note: You only need to modify the sample geometric rv() function.

Solution 3.3

(a)

```
int factorial(int n) {

    int i = 2, ret = 1;

    while(i <= n) {

        ret *= i;

        i = i + 1;

    }

    return ret;

}
```

(b)

```
#include <stdlib.h>

double rand_double() {

    /* generate random number in [0, 1) */

    double ret = (double)rand();

    return ret / (RAND_MAX + 1);

}

int sample_geometric_rv(double p) {

    double q;

    int n = 0;

    q = rand_double(); // Initialize q outside the loop

    while (q >= p) {

        q = rand_double();
```

```
        n++;

    }

    return n;

}
```

Problem 3.4

'wc' is a unix utility that displays the count of characters, words and lines present in a file. If no file is specified it reads from the standard input. If more than one file name is specified it displays the counts for each file along with the filename. In this problem, we will be implementing wc. One of the ways to build a complex program is to develop it iteratively, solving one problem at a time and testing it thoroughly. For this problem, start with the following shell and then iteratively add the missing components.

```
#include <stdio.h>
#include <stdlib.h>
int main ( int argc , char* argv [ ] )
{
FILE* fp = NULL;
int nfiles = --argc ; /* ignore the name o f the program itself */
int argidx =1; /* ignore the name of the program itself */
char* currfile="" ;
char c ;
/* count of words , lines , characters */
unsigned long nw=0, nl =0,nc=0;
i f ( n f i l e s ==0)
{
fp=s t din ; /* standard input */
n f i l e s ++;
}
el se /* s e t to f i r s t */
{
c u r r f i l e=argv [ argidx ++];
fp=fopen ( currfile , "r" );
}
while ( nfiles >0) /* files  left >0*/
{
if ( fp==NULL)
{
fprintf ( stderr , "Unable to open input\n" );
exit ( -1);
}
nc = nw = nl = 0;
while (( c = getc( fp )) != EOF)
{
/*TODO: FILL HERE
process the file using getc( fp )
*/
}
printf ( "%ld %s\n" , nc , currfile );
/* next file if exists */
nfiles --;
i f ( nfiles >0)
{
currfile = argv [ argidx ++];
fp =fopen ( currfile , "r" );
}
}
return 0 ;
}
```

Hint: In order to count words, count the transitions from non-white space to white space characters.

Solution 3.4

```c
// Solution 3.4

#include <stdio.h>

#include <stdlib.h>

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

    FILE *fp = NULL;

    int nfiles = argc - 1;

    int argidx = 1;

    char *currfile = "";

    char c;

    unsigned long nw = 0, nl = 0, nc = 0;

    if (nfiles == 0) {

        fp = stdin; // Standard input

        nfiles++;

    } else {

        currfile = argv[argidx++];

        fp = fopen(currfile, "r");

    }

    while (nfiles > 0) {

        if (fp == NULL) {

            fprintf(stderr, "Unable to open input\n");

            exit(-1);
```

```
        }

        nc = nw = nl = 0;

        int prev_char = ' ';

        while ((c = getc(fp)) != EOF) {

            nc++; // Count characters

            if (c == '\n') nl++; // Count lines

            if ((c != ' ' || c != '\t' || c != '\n') && (prev_char == ' '
|| prev_char == '\t' || prev_char == '\n')) {

                nw++;

            }

            prev_char = c;

        }

        printf("%ld %s\n", nw, currfile);

        nfiles--;

        if (nfiles > 0) {

            currfile = argv[argidx++];

            fp = fopen(currfile, "r");

        }

    }

    return 0;

}
```

Problem 3.5

In this problem, we will be reading in formatted data and generating a report. One of the common formats for interchange of formatted data is 'tab delimited' where each line corresponds to a single record. The individual fields of the record are separated by tabs. For this problem, download the file stateoutflow0708.txt from Stellar. This contains the emigration of people from individual states. The first row of the file contains the column headings. There are eight self explanatory fields. Your task is to read the file using **fscanf** and generate a report outlining the migration of people from Massachusetts to all the other states. Use the field "Aggr AGI" to report the numbers. Also, at the end, display a total and verify it is consistent with the one shown below.

An example report should look like the following:

STATE TOTAL

"FLORIDA" 590800

"NEW HAMPSHIRE" 421986

.....

Total 4609483

Make sure that the fields are aligned.

Code listing for Problem 3.1: prob1.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define NMAX 25
static unsigned int resultsbuffer[NMAX] ;
void fibonacci ()
{
/* here are the variables to set as registers */
unsigned int a = 0 ;
unsigned int b = 1 ;
unsigned int c ;
int n ;
/* do not edit below this line */
resultsbuffer[ 0 ] = a ;
resultsbuffer[ 1 ] = b ;
for (n = 2 ; n < NMAX; n++) {
c = a + b ;
resultsbuffer[ n ] = c ; /* store code in results buffer */
a = b ;
b = c ;
}
}
int main ( void) {
int n , ntests = 10000000;
clockttstart , tend ;
double favg ;
/* do p r o f i l i n g */
tstart = clock ();
for (n = 0 ; n < ntests ; n++)
fibonacci ();
tend = clock ();
/* end p r o f i l i n g */
/* compute average exe cuti on time */
favg = (( double )( tend − t s t a r t )) /CLOCKS PER SEC/ n t e s t s ;
/* print avg execution time in milli seconds */
printf ( "Avg. execution time: %g msec\n" , favg * 1 0 0 0 );
return 0 ;
}
```

Solution 3.5

```c
#include <stdio.h>

#define MAX_STATES 50

#define MAX_STATE_NAME 20

typedef struct {

    char state[MAX_STATE_NAME];

    unsigned int aggrAGI;

} StateData;

int main() {

    FILE *file = fopen("stateoutflow0708.txt", "r");

    if (file == NULL) {

        perror("Error opening file");

        return 1;

    }

    StateData states[MAX_STATES];

    int numStates = 0;

    // Skip the header line

    fscanf(file, "%*s %*s %*s %*s %*s %*s %*s %*s");

    // Read data into array

          while   (fscanf(file,   "%19s   %*s   %*s   %*s   %*s   %*s   %*s   %u",
states[numStates].state, &states[numStates].aggrAGI) == 2) {

        numStates++;

    }
```

```
    fclose(file);

    // Print the report

    printf("%-20s %-10s\n", "STATE", "TOTAL");

    for (int i = 0; i < numStates; i++) {

        printf("\"%-19s\" %-10u\n", states[i].state, states[i].aggrAGI);

    }

    // Calculate and print total

    unsigned int total = 0;

    for (int i = 0; i < numStates; i++) {

        total += states[i].aggrAGI;

    }

    printf("Total %-10u\n", total);

    return 0;

}
```

Problem Set 4

Pointers. Arrays. Strings. Searching and sorting algorithms.

Problem 4.1

Consider the insertion sort algorithm described in Lecture 5 In this problem, you will re-implement the algorithm using pointers and pointer arithmetic.

(a) The function shift element() takes as input the index of an array element that has been determined to be out of order. The function shifts the element towards the front of the array, repeatedly swapping the preceding element until the out-of-order element is in the proper location. The implementation using array indexing is provided below for your reference:

```
/* move previous elements down until insertion point reached */
void shiftelement (unsigned int i ) {
int i v a l u e ;
/* guard a g a ins t going out s ide ar ray */
for ( i v a l u e = a r r [ i ] ; i && a r r [ i -1] > i v a l u e ; i --)
a r r [ i ] = a r r [ i -1]; /* move element down */
a r r [ i ] = i v a l u e ; /* i n s e r t element */
}
```

Re-implement **this** function **using** pointers **and** pointer arithmetic instead of array indexing.

```
/* i n t *pElement - po int e r to the element
in a r r ( type i n t [ ] ) that i s out-of-pl a c e */
void s h i ft e l e m e n t ( int *pElement) {
/* i n s e r t code he r e */
}
```

Solution 4.1 (a):

```
void shift_element(int *pElement) {
    int value = *pElement;
    while (pElement > arr && *(pElement - 1) > value) {
        *pElement = *(pElement - 1);
        pElement--;
    }
    *pElement = value;
}
```

(b) The function insertion sort() contains the main loop of the algorithm. It iterates through elements of the array, from the beginning, until it reaches an element that is out-of-order. It calls shift element() to shift the offending element to its proper location earlier in the array and resumes iterating until the end is reached. The code from lecture is provided below:

```
/* it e r a t e u n ti l out-of-orde r element found ;
s h i ft the element , and continue it e r a ti n g */
void insertionsort ( void) {
unsigned int i , l en = a r r a y l e n g t h ( a r r );
for ( i = 1 ; i < l en ; i++)
i f ( a r r [ i ] < a r r [ i -1])
s h i ft e l e m e n t ( i );
}
```

Re-implement this function using pointers and pointer arithmetic instead of array indexing. Use the shift element() function you implemented in part (a).

Solution 4.1 (b):

```
void insertionsort() {
    int len = array_length(arr);
    int *ptr = arr + 1;
    for (; ptr < arr + len; ptr++) {
        if (*ptr < *(ptr - 1)) {
            shift_element(ptr - arr);
        }
    }
}
```

Problem 4.2

In this problem, we will use our knowledge of strings to duplicate the functionality of the C standard library's strtok() function, which extracts "tokens" from a string. The string is split using a set of delimiters, such as whitespace and punctuation. Each piece of the string, without its surrounding delimiters, is a token. The process of extracting a token can be split into two parts: finding the beginning of the token (the first character that is not a delimiter), and finding the end of the token (the next character is a delimiter). The first call of strtok() looks like this:

```
char * strtok(char * str, const char * delims);
```

The string str is the string to be tokenized, delims is a string containing all the single characters to be used as delimiters (e.g. " \t\r\n"), and the return value is the first token in str. Additional tokens can be obtained by calling strtok() with NULL passed for the str argument:

```
char * strtok(NULL, const char * delims);
```

Because strtok() uses a static variable to store the pointer to the beginning of the next token, calls to strtok() for different strings cannot be interleaved. The code for strtok() is provided below:

```
char * strtok( char * text , const char * de lims ) {
/* i n i t i a l i z e */
i f (! t ext )
t ext = pnexttoken ;
/* fi n d s t a r t o f token in t ext */
t ext += s t r spn ( text , de lims );
i f (* t ext == '\0' )
return NULL;
/* fi n d end o f token in t ext */
pnexttoken = t ext + s t r c spn ( text , de lims );
/* i n s e r t null-t e rminator at end */
i f (* pnexttoken != '\0' )
* pnexttoken++ = '\0' ;
return t ext ;
}
```

(a) In the context of our string tokenizer, the function strspn() computes the index of the first non-delimiter character in our string. Using pointers or array indexing (your choice), implement the strspn() function. In order to locate a character in another string, you may use the function strpos(), which is declared below:

```
int strpos ( const char * str , const char ch );
```

This function returns the index of the first occurrence of the character ch in the string str, or -1 if ch is not found. The declaration of strspn() is provided below:

```
unsigned int s t r spn ( const char * s t r , const char * de lims ) {
/* i n s e r t code he r e */
}
```

Here, delims is a string containing the set of delimiters, and the return value is the index of the first non-delimiter character in the string str. For instance, strspn(" . This", " .") == 3. If the string contains only delimiters, strspn() should return the index of the null-terminator ('\0'). Assume '\0' is not a delimiter.

Solution 4.2 (a):

```
unsigned int strspn(const char* str,const char* delims) {
    int a = strlen(str);
    int i;
    for(i=0;i<a;i++){
        if(str[i] != *delims){
            return i;
```

```
        }
    }
    return -1;
}
```

(b) The function strcspn() computes the index of the first delimiter character in our string. Here's the declaration of strcspn():

```
unsigned int s t r c spn ( const char * s t r , const char * de lims ) {
/* i n s e r t code he r e */
}
```

If the string contains no delimiters, return the index of the null-terminator ('\0'). Implement this function using either pointers or array indexing.

Solution 4.2 (b):

```
unsigned int strcspn(const char* str,const char*  delims){
    int a=strlen(str);
    int i;
    for(i=0;i<a;i++){
        if(str[i] == *delims){
            return i;
        }
    }
    return 0;
}
```

Problem 4.3

In this problem, you will be implementing the shell sort. This sort is built upon the insertion sort, but attains a speed increase by comparing far-away elements against each other before comparing closer-together elements. The distance between elements is called the "gap". In the shell sort, the array is sorted by sorting gap sub-arrays, and then repeating with a smaller gap size. As written here, the algorithm sorts in O(n2) time. However, by adjusting the sequence of gap sizes used, it is possible to improve the performance to O(n3/2), O(n4/3), or even O(n(logn)2) time. You can read about a method for performing the shell sort in O(n(log n)2) time on Robert Sedgewick's page at Princeton: http://www.cs.princeton.edu/~rs/shell/paperF.pdf

Note that you can find complete C code for the shell sort at the beginning of the paper, so please wait until you have finished this exercise to read it.

(a) First, we will modify the shift element() function from the insertion sort code for the shell sort. The new function, started for you below, should start at index i and shift by intervals of size gap. Write the new function, using array indexing or pointers. Assume that i ≥ gap.

```
void shift e l e m e n t b y g a p (unsigned int i , unsigned int gap ) {
/* i n s e r t code he r e */
}
```

Solution 4.3 (a):

```
void shiftelementbygap(unsigned int i, unsigned int gap) {
    int temp = arr[i];
    int j;
    for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
        arr[j] = arr[j - gap];
    }
    arr[j] = temp;
}
```

(b) Now, we need to write the main shell sort routine. Using the template below, fill in the missing code so that the insertion sort in the inner loop compares every element to the element gap spaces before it.

```
void s h e l l s o r t ( void) {
unsigned int gap , i ,
l en = a r r a y l e n g t h ( a r r );
/* sor t , comparing a g a ins t f a r t h e r away
e l ement s f i r s t , then c l o s e r e l ement s */
for ( gap = l en / 2 ; gap > 0 ; gap /= 2) {
/* do i n s e r ti o n −l i k e sor t , but comparing
and s h i fti n g e l ement s in m u lti p l e s o f gap */
for ( /* i n s e r t code he r e */ ) {
i f ( /* i n s e r t code he r e */ ) {
/* out o f order , do s h i ft */
s h ift e l e m e n t b y g a p ( i , gap );
}
}
}
}
```

Solution 4.3 (b):

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

```
void shiftelementbygap(unsigned int i, unsigned int gap);
void shellsort();
void display();

int arr[7] = {1, 2, 4, 3, 8, 7, 5};

int main() {
    shellsort();
    display();
    return 0;
}

void shiftelementbygap(unsigned int i, unsigned int gap) {
    int temp = arr[i];
    int j;
    for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
        arr[j] = arr[j - gap];
    }
    arr[j] = temp;
}

void shellsort() {
    unsigned int len = 7;
    for (unsigned int gap = uint(len / 2); gap > 0; gap /= 2) {
        for (unsigned int i = gap; i < len; i += 1) {
            shiftelementbygap(i, gap);
        }
    }
}

void display() {
    for (int i = 0; i < 7; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```