Name: Section: Rollno:

ESC101A: Fundamentals of Computing (End Semester Exam - A)

21st Nov, 2014

Total Number of Pages: 25 Total Points 200

Instructions: Read Carefully.

1. Write you roll number on all the pages of the answer book.
2. Write the answers cleanly in the space provided. There is space left on the back of the answer book for rough work.
3. Do not exchange question books or change the seat after obtaining question paper.
4. **Use pens (blue/black ink) and not pencils** for answering. Also do not use red pens.
5. Even if no answers are written, the answer book has to be returned back with name and roll number written.

Helpful hints

1. The questions are *not* arranged according to the increasing order of difficulty. Do a quick first round where you answer the easy ones and leave the difficult ones for the subsequent rounds.
2. All blanks may NOT carry equal marks.
3. Use the cheat sheet provided in the answer book for any doubt related to C programming (Not all topics in the cheat sheet are covered in class, Not all topics covered in the class are provided in the cheat sheet.)

Question	Points	Score
1	30	
2	25	
3	15	
4	20	
5	20	
6	30	
7	15	
8	25	
9	20	
Total:	200	

I pledge my honour as a gentleman/lady that during the examination I have neither given assistance nor received assistance.

Signature

Question 1. (30 points) The table on the next page contains program fragments in C language in column L. You have to match each of the fragment to all its equivalent behaviors (expressions or the expected values) in column R.

Note that **more than one** expression in column L can match a behavior in column R. Also, same expression in column L can match **more than one** behavior in column R. You have to list **ALL** matching behaviors in the space provided.

Assume all the variables used are integers, and they hold small integer values (including zero), so that no overflow (or wrap-around) occurs.

Give your answer here:

L	**R**
1.	E
2.	H
3.	N
4.	A, D
5.	B
6.	L
7.	N
8.	M
9.	L
10.	E

NOTE: Option R.C is NOT same as L.1 or L.10; R.C is a STATEMENT, while L.1 and L.10 are EXPRESSIONS. You can say c = *(&a), c = a ? a : 0 BUT NOT c = if (a==0) a else 0;

Column L		Column R	

1.	a ? a : 0	A.	if (a) c = 1;
			else if (b) c = 1;
			else c = 0;

2.	a = a * ++b;	B.	if (a) c = !(!b);
			else c = 0;

3.	a = 0 && ++b;	C.	if (a == 0) 0;
			else a;

4.	c = a || b;	D.	if (!b) c = a ? 1 : 0;
			else c = 1;

5.	c = a && b;	E.	a
6.	&(*a)	F.	a^2

7.	c = (b=0) && a;	G.	b = b + 1;
			a = a * (b + 1);

8.	(a*2) | | 1	H.	a = a * (b + 1);
			b = b + 1;

9.	a = a * ++1;	I.	2*a
10.	*(&a)	J.	0 (ZERO)
		K.	Undefined
		L.	Syntax Error
		M.	1 (ONE)
		N.	Defined, but none of the above

Question 2. (25 points) For each of the programs below, write the output generated by the program. (Each program has only **one** printf call, but the printf can print **more than one** values.) There are **FIVE** programs (A, B, C, D, E).

(A)

```c
#include<stdio.h>
int main()
{
    char *s[] = {"black", "white", "pink", "violet"};
    char **ptr[4], ***p;
    int i;

    for (i=3; i >=0; i--)
        ptr[3-i] = s+i;

    p = ptr;
    ++p;

    printf("%s", **p+1);
    return 0;
}
```

OUTPUT (A): [6 points]

(B)

```c
#include<stdio.h>
int main()
{
    int i=3, *j, k;
    j = &i;
    k = *j;
    i = 4;
    k = k + i * *j * i;
    printf("%d", k);
    return 0;
}
```

OUTPUT (B): [3 points]

(C)

```c
#include<stdio.h>
int main()
{
    char str[20] = "Hello";
    char *p=str;
    *p=’M’;
    printf("%s", str);
    return 0;
}
```

OUTPUT (C): [2 points]

(D)

```
#include<stdio.h>
int main()
{
    int arr[2][2][2] = {{{8, 7}, {6, 5}}, {{4, 3}, {2, 1}}};
    int *p, *q;
    p = &arr[1][0][1];
    q = (int*) arr;
    printf("%d, %d\n", *p, *q);
    return 0;
}
```

OUTPUT (D): [4 + 3 points]

(E)

```
#include<stdio.h>
int *check(int* i, int j)
{
    int *p, *q;
    p = i;
    *p = *p + 10;
    q = &j;
    *q = *q * *p;
    return q;
}

int main()
{
    int p=5, *c;
    c = check(&p, p);
    printf("%d, %d", *c, p);
    return 0;
}
```

OUTPUT (E): [3 + 4 points]

Solution: A: ink
B: 67
C: Mello
D: 3, 8
E: Undefined , 15
(Undefined, garbage, unknown,... anything is OK, but not 75 :-))

Question 3. (15 points) The following program reverses a linked list *in-place* using constant amount of extra space. No node is created or destroyed in the process.

Complete the program so that it works as desired.

```
/* structure of linked list */
typedef struct elem{
    int data;
    struct elem* next;
} Element;

/* Given the head of a linked list as input, reverses the linked list.
 * Returns the head of the reversed linked list.
 */
Element *reverse(Element *head)
{
    Element *prev = NULL;
    Element *curr = head;
    Element *next = NULL;
    while (curr != NULL) {

        next         = ___________;

        curr->next   = ___________;

        ___________  = ___________;

        curr         = next;
    }

    return ___________;
}
```

Solution:

```
/* structure of linked list */
typedef struct elem{
    int data;
    struct elem* next;
} Element;

/* Given the head of a linked list as input, reverses the linked
    list.
 * Returns the head of the reversed linked list.
 */
Element *reverse(Element *head)
{
    Element *prev = NULL;
    Element *curr = head;
    Element *next = NULL;
    while (curr != NULL) {
        next = head->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}
```

Question 4. (20 points) Find the output of the the following program for the given set of inputs. (**One printf** on line 38).

```
1  #include <stdio.h>
2  #define MAX_SIZE 20
3  int main()
4  {
5      int n,m,i;
6      int a[MAX_SIZE];
7      scanf("%d%d",&n,&m);
8      for(i=0;i<n;i++)
9          scanf("%d",&a[i]);
10     int min_index=0, len=1, sum=a[0], ans=0;
11
12     for(i=1;i<n;i++)
13     {
14         if(a[i]+sum<=m)
15         {
16             sum = sum+a[i];
17             len++;
18         }
19         else
20         {
21             sum=sum+a[i];
22             len++;
23             while(sum>m && min_index<=i)
24             {
25                 sum = sum - a[min_index];
26                 min_index++;
27                 len--;
28             }
29             if(min_index>i)
30             {
31                 sum = 0;
32                 len = 0;
33             }
34         }
35         if(sum > ans)
36                 ans = sum;
37     }
38     printf("%d\n",ans);
39 }
```

INPUT	INPUT
4 9 7 3 5 6	5 16 6 6 8 5 9
OUTPUT	**OUTPUT**

Solution: 8 14

Question 5. (20 points) Find the output of the the following program for the given set of inputs. (**2 printf**s on lines 42 and 43).

```
 1 #include <stdio.h>
 2 #define SIZE 4
 3 
 4 void f1(int a[][SIZE], int x1, int x2, int y1, int y2){
 5     int t = a[x1][x2];
 6     a[x1][x2] = a[y1][y2];
 7     a[y1][y2] = t;
 8 }
 9 
10 void f2(int a[][SIZE]){
11     int i,j;
12     for (i = 0; i < SIZE; ++i){
13         for (j = 0; j < i; ++j){
14             f1(a,i,j,j,i);
15         }
16     }
17 }
18 
19 void f3(int a[][SIZE]){
20     int i,j;
21     for (i = 0; i < SIZE/2; ++i){
22         for (j = 0; j < SIZE; ++j){
23             f1(a,i,j,SIZE-i-1,j);
24         }
25     }
26 }
27 
28 int main(){
29     int a[SIZE][SIZE];
30     int i,j;
31 
32     // Taking Input.
33     for (i = 0; i < SIZE; ++i)
34         for (j = 0; j < SIZE; ++j)
35             scanf("%d",&a[i][j]);
36 
37     f2(a);
38     f3(a);
39 
40     for (i = 0; i < SIZE; ++i){
41         for (j = 0; j < SIZE-1; ++j)
42             printf("%d ", a[i][j]);      // <<<=== PRINTF
43         printf("%d\n", a[i][SIZE-1]);    // <<<=== PRINTF
44     }
45     return 0;
46 }
```

INPUT

1 1 1 1

2 4 8 16

3 9 27 51

4 16 64 216

OUTPUT

Solution: The code rotates SIZExSIZE matrix 90 deg anti-clockwise.

Question 6. (30 points) The program given below is a partially filled code which takes names of three files, say `src`, `dest1` and `dest2`, as **command line arguments**. It then reads the file `src` and copies its lines alternately to `dest1` and `dest2` (i.e., lines 1,3,5,... from `src` go to `dest1` and lines 2,4,6,... go to `dest2`.

For example, Suppose we have the following lines in a file named **PlayerInfo.txt**

```
Sachin Tendulkar
Mumbai
M S Dhoni
Jharkhand
Virat Kohli
Delhi
Rahul Dravid
Karnataka
Bhuvaneshwar Kumar
U.P.
Akshar Patel
Gujrat
```

then, after the following command is executed,

```
./a.out PlayerInfo.txt PlayerName.txt PlayerState.txt
```

we get two (possibly new) files **PlayerName.txt** and **PlayerState.txt**

PlayerName.txt

```
Sachin Tendulkar
M S Dhoni
Virat Kohli
Rahul Dravid
Bhuvaneshwar Kumar
Akshar Patel
```

PlayerState.txt

```
Mumbai
Jharkhand
Delhi
Karnataka
U.P.
Gujrat
```

Complete the following program to do the given task.

```
#include<stdio.h>

int main(int argc, char **argv)
{
  FILE* fread; /*file to read from*/
  FILE* fwrite[2]; /*files to write to*/

  if(____________________) {
    printf("Incorrect number of arguments\n");
    return 0;
  }

  /*handle command line arguments to open files*/

  fread = fopen(____________, ____________ ); /*1st arg: file to read*/

  fwrite[0] = fopen(____________, ____________ ); /*2nd: file to write*/

  fwrite[1] = fopen(____________, ____________ ); /*3rd: file to write*/
  /*error checking related to file opening ignored.*/

  int widx = 0; // index of FILE* of file we are writing to.

  while(____________________){ /*still stuff to read.*/
    char c;
    fscanf(fread, "%c", &c);     /*read a character*/

    fprintf(____________________, ____________, ____________);  /*write to proper file*/

    if (c=='\n') { /*switch the output file*/
      widx = 1 - widx; /*0->1, 1->0*/
    }
  }

  /*cleanup: close the files*/
  fclose(fread);

  fclose(____________________);

  fclose(____________________);

  return 0;
}
```

Solution:

```
#include<stdio.h>

int main(int argc, char **argv)
{
    FILE* fread; /* file to read from */
    FILE* fwrite[2]; /* files to write to */

    if(argc != 4) {
        printf("Incorrect number of arguments\n");
        return 0;
    }

    fread = fopen(argv[1], "r" );
    fwrite[0] = fopen(argv[2], "w" );
    fwrite[1] = fopen(argv[3], "w" );
    /* error checking related to file opening ignored. */

    int widx = 0; // index of FILE* of file we are writing to.
    while(!feof(fread)){ /* still stuff to read. */
        char c;
        fscanf(fread, "%c", &c);         /* read a charcter */
        fprintf(fwrite[widx], "%c", c);  /* write to proper file */

        if (c==’\n’) { /* switch the output file */
            widx = 1 - widx; /* 0->1, 1->0 */
        }
    }

    /* cleanup: close the files */
    fclose(fread);
    fclose(fwrite[0]);
    fclose(fwrite[1]);

    return 0;
}
```

Question 7. (15 points) Given a a **binary tree** of integer data values and an integer **sum**, we want to determine if the tree has a path such that the data values along the path add up to the given sum. The path must be a root-to-leaf path, that is starting at root and ending at a node whose both children are NULL.

for example, consider the tree,

```
          5
         / \
        4   8
       /   / \
      11  13  4
     /  \      \
    7    2      1
```

[In picture, the integer denotes the data value stored in a node (e.g. 5) , and the two lines (/ and \ ) show the left and right child respectively. NULL children are not shown.]

- For sum=22, there exists a root-to-leaf path 5→4→11→2 for which sum is 22.
- For sum=7, there is no root-to-leaf path having sum 7.

Fill in the function which given a pointer to the **root** of the binary tree and the **sum**, returns 1 if there exists a root-to-leaf path whose sum matches the input else returns 0.

```
/* Definition for binary tree */
typedef struct _TreeNode {
    int val;
    struct _TreeNode *left;
    struct _TreeNode *right;
} TreeNode;

int hasPathSum(TreeNode *root, int sum)
{
    if (___________) /* base case: no tree! */
        return 0;

    /* take the value in root into account */
    int remaining = sum - root->val;

    /* no child => leaf */
    if ((root->left == NULL) && (root->right == NULL))
        return remaining == ___________;

    /* Only right child : search on right */
    if ((root->left == NULL) && (__________________))
        return hasPathSum(root->right, remaining);

    /* Only left child : search on left  */
    if ((__________________) && (root->right == NULL))
        return hasPathSum(root->left, remaining);

    /* Both children : search on both sides  */
    return ______________________________________________________;
}
```

Solution:

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

/* Definition for binary tree */
typedef struct _TreeNode {
    int val;
    struct _TreeNode *left;
    struct _TreeNode *right;
} TreeNode;

int hasPathSum(TreeNode *root, int sum)
{
    if (root==NULL)
        return 0;

    /* take the value in root into account */
    int remaining = sum - root->val;

    /* no child => leaf */
    if ((root->left == NULL) && (root->right == NULL))
        return remaining == 0;

    /* Only right child : search on right */
    if ((root->left == NULL) && (root->right != NULL))
        return hasPathSum(root->right, remaining);

    /* Only left child : search on left  */
    if ((root->left != NULL) && (root->right == NULL))
        return hasPathSum(root->left, remaining);

    /* Both children : search on both sides  */
    return hasPathSum(root->left, remaining) || hasPathSum(root->right, remaining);
}

TreeNode* mkNode(int val, TreeNode* left, TreeNode* right)
{
    TreeNode* nd = (TreeNode*) malloc(sizeof(TreeNode));
    nd->val   = val;
    nd->left  = left;
    nd->right = right;

    return nd;
}

int main()
{
    int sum;
    scanf ("%d", &sum);
    TreeNode *root = mkNode(5,
                            mkNode(4,
                                   mkNode(11,
                                          mkNode(7, 0, 0),
                                          mkNode(2, 0, 0)),
                                   0),
                            mkNode(8,
                                   mkNode(13, 0, 0),
                                   mkNode(4,
                                          0,
                                          mkNode(1, 0, 0))));

    printf("?? : %d\n", hasPathSum(root, sum));
    return 0;

}
```

Question 8. (25 points) Peter is given 2 strings, s1 and s2. He wants to find the minimum number of changes he should do to convert string s1 into s2. A change can be one of the following:

- Insert one character at any location
- Delete one character at any location
- Replace a character by any other character

He calls the *number of changes* required as edit cost.

Peter is a big fan of recursive functions, so he tries to solve the problem using recursion. Let s(j) denotes the string formed by first j characters of string s (i.e., characters in the array s[0..j-1]).

To calculate the edit cost of s1(n) and s2(m), Peter comes up with the following *recursive* cases to convert s1(n) to s2(m):

Case d1: convert s1(n-1) to s2(m) and delete n^{th} character of s1.

Case d2: convert s1(n) to s2(m-1) and add m^{th} character of s2.

Case d3: convert s1(n-1) to s2(m-1); compare n^{th} character of s1 with m^{th} character of s2. if they are not same, replace n^{th} character of s1 with m^{th} character of s2.

The desired edit cost is the minimum of the costs of the three cases.

Help Peter get the required edit cost from d1, d2 and d3 by filling out the blanks in the code on the next page.

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

int get_minimum (int a, int b, int c) {
   if (a <= b && a <= c) {
      return a;
   }
   if (b <= a && b <= c) {
      return b;
   }
   return c;
}
/* compute minimum edit cost for s1[0...n-1] and s2[0...m-1] */
int EditCost (char *s1, char *s2, int n, int m) {
   if (n == 0 && m == 0) {
      return 0;
   }
   if (n == 0) {

      return ___________; /* 1 blank */
   }
   if (m == 0) {

      return ___________; /* 1 blank */
   }

   int d1, d2, d3;
   d1 = EditCost(___________, ___________, ___________, ___________)+1; /*4 blanks*/
   d2 = EditCost(___________, ___________, ___________, ___________)+1; /*4 blanks*/
   d3 = EditCost(___________, ___________, ___________, ___________); /*4 blanks*/

   if (______________________) { /* 1 blank */
      d3 = d3 + 1;
   }

   return get_minimum(d1, d2, d3);
}

int main () {
   char s1[100], s2[100];
   int edit_cost;
   scanf("%s%s", s1, s2);
   edit_cost = EditCost(s1, s2, strlen(s1), strlen(s2));
   printf("Edit cost between %s and %s is %d.\n", s1, s2, edit_cost);
   return 0;
}
```

Solution:

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

int get_minimum (int a, int b, int c) {
    if (a <= b && a <= c) {
       return a;
    }
    if (b <= a && b <= c) {
       return b;
    }
    return c;
}

int EditCost (char *s1, char *s2, int n, int m) {
    if (n == 0 && m == 0) {
       return 0;
    }
    if (n == 0) {
       return m;                                   // Blank  (1)
    }
    if (m == 0) {
       return n;                                   // Blank  (2)
    }

    int d1, d2, d3;
    d1 = EditCost(s1, s2, n-1, m) + 1;  // Blanks (3), (4), (5), (6)
    d2 = EditCost(s1, s2, n, m-1) + 1;  // Blanks (7), (8), (9),
        (10)
    d3 = EditCost(s1, s2, n-1, m-1);    // Blanks (11),(12),(13)
        ,(14)

    if (s1[n-1] != s2[m-1]) {                      // Blank  (15)
       d3 = d3 + 1;
    }

    return get_minimum(d1, d2, d3);
}

int main () {
    char s1[100], s2[100];
    int edit_cost;
    scanf("%s%s", s1, s2);
    edit_cost = EditCost(s1, s2, strlen(s1), strlen(s2));
    printf("Edit cost for %s and %s is %d.\n", s1, s2, edit_cost);
    return 0;
}
```

Question 9. (20 points) A regular expression (regex for short) is a string of characters to describe search patterns. A **text** string is matched against the regular expression **pattern** to decide if the text can be generated by the pattern or not. Assume simple regular expressions where the pattern comprises of alphabet symbols (a. . . z for our problem) along with two special symbols * and +. The meaning of * and + in the pattern is as follows:

- The character occurring before a * in the pattern can be matched zero or more times in the text.
- The character occurring before a + in the pattern can be matched one or more times in the text.

Our patterns will not have consecutive special characters (* or +). Thus,

Pattern **"a"** matches text **"a"** only.

Pattern **"a*"** matches text **""** **(empty string), "a", "aa", "aaa", . . .** (any number of **a**-s).

Pattern **"a+"** matches text **"a", "aa", "aaa", . . .** (any number of **a**-s).

Patterns **"a+*" "a++" "a**"** etc. are invalid.

Some more examples of valid patterns and the strings that can be generated from such patterns are:

Pattern	**Strings that match**
"a*bba+"	"bba", "bbaa", "abba", "aabbaaaa" etc.
"a*b*a+"	"a", "aa", "aba", "ba", "bbbba", "bbbbaaa" etc.
"abc*b+c"	"abcbc", "abbc", "abcbbbc", "abcccbc" etc.

The following code implements function **int match(char *text, char *pattern)** to match a given regex **pattern** to a given **text** string. The function returns **1** (true) if the string in text matches the regular expression in pattern string, otherwise it generates **0** (false).

Call	**Return Value**
match("bba", "a*bba+")	1
match("bb", "a*bba+")	0
match("abccbc", "abc*b+c")	1
match("abbcc", "abc*b+c")	0

Complete the code so that it does the job of matching.

```
int match(char *text, char *pattern)
{
    if(*text=='\0'){ /*text fully read*/

        if (______________________) /*pattern fully read => MATCH.*/
            return 1;
        /*text finished, but pattern remaining*/
        /*If the the NEXT character in pattern is *, skip the CURRENT
            character along with *. */
        if (*(pattern+1)=='*')
            return match(text,pattern+2);
        /*If the CURRENT character is a +, pattern should be finished
            after skipping it.*/
        if (*pattern == '+')
            return  __________________ == '\0';
        /*At this point, current character in pattern is not followed
            by a '*' or a '+', so we have a MISMATCH.*/
        return 0;
    }
    /*We have text remaining.*/
    if(*(pattern+1)=='*')
    {
        /*The next character in the pattern is a *, we can either match
             OR ignore the current character (with *) in the text.*/
        if(*pattern==*text)
            return match(text+1,pattern) ||

                    match(__________________,__________________);
        else /*we are forced to ignore current character (with *)*/

            return match(text,__________________);
    }
    if (*pattern == '+'){
        if (*text==*(pattern-1))
            /*If current pattern character is +, either we match the
                current text character with the pattern character
                preceding +, or try to match after skipping it.*/

            return match(text+1, __________________) ||

                     match(text, __________________);
        else
            return match(text, __________________);
    }
    if(*text==*pattern)
        /*character in text matches character in pattern*/
        return match(text+1,pattern+1);
    return 0;
}
```

Solution:

```
#include<stdio.h>
#define true 1
#define false 0
int match(char *text, char *pattern)
{
        printf("%c, %c\n", *text, *pattern);
        if(*text=='\0'){
                //If both the pattern and text are fully read,
                // they have matched.
                if (*pattern=='\0')
                        return true;
                //If the the next character in pattern is *
                //skip the current character along with star.
                if (*(pattern+1)=='*')
                        return match(text,pattern+2);
                //If the current character is a +, we can ignore
                //the +, as we have no characters in text.
                if (*pattern == '+')
                        return *(pattern+1) == '\0';
                //If the current character in pattern is not followed
                //by a *, the pattern mismatches the text.
                else
                        return false;
        }
        if(*(pattern+1)=='*')
        {
                //If the next character in the pattern is a *,
                // we can either match or ignore the current character in the text.
                if(*pattern==*text)
                        return match(text+1,pattern) || match(text,pattern+2);
                else
                        return match(text,pattern+2);
        }
        if (*pattern == '+'){
                if (*text==*(pattern-1))
                        //If current pattern character is +, either we match the current text
                        //character with the pattern character preceding +,
                        //or try to match after skipping it.
                        return match(text+1, pattern) || match(text, pattern+1);
                else
                        return match(text, pattern+1);
        }
        if(*text==*pattern)
                //character in text matches character in pattern
                return match(text+1,pattern+1);

        return false;
}

int main(int argc, char**argv)
{
        char *text=argv[1];
        char *pattern=argv[2];
        printf("%d\n", (match(text,pattern) ? 1 : 0));
        return 0;
}
```

C Reference Card (ANSI)

Program Structure/Functions

type fnc(*type*$_1$, ...);	function prototype
type name;	variable declaration
int main(void) {	main routine
declarations	local variable declarations
statements	
}	
type fnc(*arg*$_1$, ...) {	function definition
declarations	local variable declarations
statements	
return *value*;	
}	
/* */	comments
int main(int argc, char *argv[])	main with args
exit(*arg*);	terminate execution

C Preprocessor

include library file	#include <*filename*>
include user file	#include "*filename*"
replacement text	#define *name text*
replacement macro	#define *name*(*var*) *text*
Example. #define max(A,B) ((A)>(B) ? (A) : (B))	
undefine	#undef *name*
quoted string in replace	#
Example. #define msg(A) printf("%s = %d", #A, (A))	
concatenate args and rescan	##
conditional execution	#if, #else, #elif, #endif
is *name* defined, not defined?	#ifdef, #ifndef
name defined?	defined(*name*)
line continuation char	\

Data Types/Declarations

character (1 byte)	char
integer	int
real number (single, double precision)	float, double
short (16 bit integer)	short
long (32 bit integer)	long
double long (64 bit integer)	long long
positive or negative	signed
non-negative modulo 2^m	unsigned
pointer to int, float,...	int*, float*,...
enumeration constant	enum *tag* {*name*$_1$=*value*$_1$,...};
constant (read-only) value	*type* const *name*;
declare external variable	extern
internal to source file	static
local persistent between calls	static
no value	void
structure	struct *tag* {...};
create new name for data type	typedef *type name*;
size of an object (type is size_t)	sizeof *object*
size of a data type (type is size_t)	sizeof(*type*)

Initialization

initialize variable	*type name*=*value*;
initialize array	*type name*[]={*value*$_1$,...};
initialize char string	char *name*[]="*string*";

Constants

suffix: long, unsigned, float	65536L, -1U, 3.0F
exponential form	4.2e1
prefix: octal, hexadecimal	0, 0x or 0X
Example. 031 is 25, 0x31 is 49 decimal	
character constant (char, octal, hex)	'a', '\ooo', '\x*hh*'
newline, cr, tab, backspace	\n, \r, \t, \b
special characters	\\\\, \?, \\', \\"
string constant (ends with '\0')	"abc...de"

Pointers, Arrays & Structures

declare pointer to *type*	*type* **name*;
declare function returning pointer to *type*	*type* *f();
declare pointer to function returning *type*	*type* (*pf)();
generic pointer type	void *
null pointer constant	NULL
object pointed to by *pointer*	**pointer*
address of object *name*	&*name*
array	*name*[*dim*]
multi-dim array	*name*[*dim*$_1$][*dim*$_2$]...

Structures

```
    struct tag {          structure template
      declarations        declaration of members
    };
```

create structure	struct *tag name*
member of structure from template	*name*.*member*
member of pointed-to structure	*pointer* -> *member*
Example. (*p).x and p->x are the same	
single object, multiple possible types	union
bit field with *b* bits	unsigned *member*: *b*;

Operators (grouped by precedence)

struct member operator	*name*.*member*
struct member through pointer	*pointer*->*member*
increment, decrement	++, --
plus, minus, logical not, bitwise not	+, -, !, ~
indirection via pointer, address of object	**pointer*, &*name*
cast expression to type	(*type*) *expr*
size of an object	sizeof
multiply, divide, modulus (remainder)	*, /, %
add, subtract	+, -
left, right shift [bit ops]	<<, >>
relational comparisons	>, >=, <, <=
equality comparisons	==, !=
and [bit op]	&
exclusive or [bit op]	^
or (inclusive) [bit op]	|
logical and	&&
logical or	||
conditional expression	*expr*$_1$? *expr*$_2$: *expr*$_3$
assignment operators	+=, -=, *=, ...
expression evaluation separator	,

Unary operators, conditional expression and assignment operators group right to left; all others group left to right.

Flow of Control

statement terminator	;
block delimiters	{ }
exit from switch, while, do, for	break;
next iteration of while, do, for	continue;
go to	goto *label*;
label	*label*: statement
return value from function	return *expr*

Flow Constructions

```
if statement          if (expr_1)  statement_1
                      else if (expr_2)  statement_2
                      else  statement_3
while statement       while (expr)
                        statement
for statement         for (expr_1; expr_2; expr_3)
                        statement
do statement          do  statement
                      while(expr);
switch statement      switch (expr) {
                         case const_1: statement_1 break;
                         case const_2: statement_2 break;
                         default: statement
                      }
```

ANSI Standard Libraries

<assert.h>	<ctype.h>	<errno.h>	<float.h>	<limits.h>
<locale.h>	<math.h>	<setjmp.h>	<signal.h>	<stdarg.h>
<stddef.h>	<stdio.h>	<stdlib.h>	<string.h>	<time.h>

Character Class Tests <ctype.h>

alphanumeric?	isalnum(c)
alphabetic?	isalpha(c)
control character?	iscntrl(c)
decimal digit?	isdigit(c)
printing character (not incl space)?	isgraph(c)
lower case letter?	islower(c)
printing character (incl space)?	isprint(c)
printing char except space, letter, digit?	ispunct(c)
space, formfeed, newline, cr, tab, vtab?	isspace(c)
upper case letter?	isupper(c)
hexadecimal digit?	isxdigit(c)
convert to lower case	tolower(c)
convert to upper case	toupper(c)

String Operations <string.h>

s is a string; cs, ct are constant strings

length of s	strlen(s)
copy ct to s	strcpy(s,ct)
concatenate ct after s	strcat(s,ct)
compare cs to ct	strcmp(cs,ct)
only first n chars	strncmp(cs,ct,n)
pointer to first c in cs	strchr(cs,c)
pointer to last c in cs	strrchr(cs,c)
copy n chars from ct to s	memcpy(s,ct,n)
copy n chars from ct to s (may overlap)	memmove(s,ct,n)
compare n chars of cs with ct	memcmp(cs,ct,n)
pointer to first c in first n chars of cs	memchr(cs,c,n)
put c into first n chars of s	memset(s,c,n)

C Reference Card (ANSI)

Input/Output <stdio.h>

Standard I/O

standard input stream	stdin
standard output stream	stdout
standard error stream	stderr
end of file (type is int)	EOF
get a character	getchar()
print a character	putchar(*chr*)
print formatted data	printf("*format*",arg_1,...)
print to string s	sprintf(s,"*format*",arg_1,...)
read formatted data	scanf("*format*",&$name_1$,...)
read from string s	sscanf(s,"*format*",&$name_1$,...)
print string s	puts(s)

File I/O

declare file pointer	FILE **fp*;
pointer to named file	fopen("*name*","*mode*")

modes: r (read), w (write), a (append), b (binary)

get a character	getc(*fp*)
write a character	putc(*chr*,*fp*)
write to file	fprintf(*fp*,"*format*",arg_1,...)
read from file	fscanf(*fp*,"*format*",arg_1,...)
read and store n elts to *ptr	fread(*ptr,eltsize,n,*fp*)
write n elts from *ptr to file	fwrite(*ptr,eltsize,n,*fp*)
close file	fclose(*fp*)
non-zero if error	ferror(*fp*)
non-zero if already reached EOF	feof(*fp*)
read line to string s (< max chars)	fgets(s,max,*fp*)
write string s	fputs(s,*fp*)

Codes for Formatted I/O: "%-+ 0*w.pmc*"

-	left justify
+	print with sign
space	print space if no sign
0	pad with leading zeros
w	min field width
p	precision
m	conversion character:

h short, l long, L long double

c conversion character:

d,i	integer	u	unsigned
c	single char	s	char string
f	double (printf)	e,E	exponential
f	float (scanf)	lf	double (scanf)
o	octal	x,X	hexadecimal
p	pointer	n	number of chars written

g,G same as f or e,E depending on exponent

Variable Argument Lists <stdarg.h>

declaration of pointer to arguments	va_list *ap*;
initialization of argument pointer	va_start(*ap*,*lastarg*);

lastarg is last named parameter of the function

access next unnamed arg, update pointer	va_arg(*ap*,*type*)
call before exiting function	va_end(*ap*);

Standard Utility Functions <stdlib.h>

absolute value of int n	abs(n)
absolute value of long n	labs(n)
quotient and remainder of ints n,d	div(n,d)

returns structure with div_t.quot and div_t.rem

quotient and remainder of longs n,d	ldiv(n,d)

returns structure with ldiv_t.quot and ldiv_t.rem

pseudo-random integer [0,RAND_MAX]	rand()
set random seed to n	srand(n)
terminate program execution	exit(status)
pass string s to system for execution	system(s)

Conversions

convert string s to double	atof(s)
convert string s to integer	atoi(s)
convert string s to long	atol(s)
convert prefix of s to double	strtod(s,&endp)
convert prefix of s (base b) to long	strtol(s,&endp,b)
same, but unsigned long	strtoul(s,&endp,b)

Storage Allocation

allocate storage	malloc(size), calloc(nobj,size)
change size of storage	newptr = realloc(ptr,size);
deallocate storage	free(ptr);

Array Functions

search array for key	bsearch(key,array,n,size,cmpf)
sort array ascending order	qsort(array,n,size,cmpf)

Time and Date Functions <time.h>

processor time used by program	clock()

Example. clock()/CLOCKS_PER_SEC is time in seconds

current calendar time	time()
$\texttt{time}_2$-$\texttt{time}_1$ in seconds (double)	difftime($\texttt{time}_2$,$\texttt{time}_1$)
arithmetic types representing times	clock_t,time_t
structure type for calendar time comps	struct tm

tm_sec	seconds after minute
tm_min	minutes after hour
tm_hour	hours since midnight
tm_mday	day of month
tm_mon	months since January
tm_year	years since 1900
tm_wday	days since Sunday
tm_yday	days since January 1
tm_isdst	Daylight Savings Time flag

convert local time to calendar time	mktime(tp)
convert time in tp to string	asctime(tp)
convert calendar time in tp to local time	ctime(tp)
convert calendar time to GMT	gmtime(tp)
convert calendar time to local time	localtime(tp)
format date and time info	strftime(s,smax,"*format*",tp)

tp is a pointer to a structure of type tm

Mathematical Functions <math.h>

Arguments and returned values are double

trig functions	sin(x), cos(x), tan(x)
inverse trig functions	asin(x), acos(x), atan(x)
arctan(y/x)	atan2(y,x)
hyperbolic trig functions	sinh(x), cosh(x), tanh(x)
exponentials & logs	exp(x), log(x), log10(x)
exponentials & logs (2 power)	ldexp(x,n), frexp(x,&e)
division & remainder	modf(x,ip), fmod(x,y)
powers	pow(x,y), sqrt(x)
rounding	ceil(x), floor(x), fabs(x)

Integer Type Limits <limits.h>

The numbers given in parentheses are typical values for the constants on a 32-bit Unix system, followed by minimum required values (if significantly different).

CHAR_BIT	bits in char	(8)
CHAR_MAX	max value of char	(SCHAR_MAX or UCHAR_MAX)
CHAR_MIN	min value of char	(SCHAR_MIN or 0)
SCHAR_MAX	max signed char	(+127)
SCHAR_MIN	min signed char	(−128)
SHRT_MAX	max value of short	(+32,767)
SHRT_MIN	min value of short	(−32,768)
INT_MAX	max value of int	(+2,147,483,647) (+32,767)
INT_MIN	min value of int	(−2,147,483,648) (−32,767)
LONG_MAX	max value of long	(+2,147,483,647)
LONG_MIN	min value of long	(−2,147,483,648)
UCHAR_MAX	max unsigned char	(255)
USHRT_MAX	max unsigned short	(65,535)
UINT_MAX	max unsigned int	(4,294,967,295) (65,535)
ULONG_MAX	max unsigned long	(4,294,967,295)

Float Type Limits <float.h>

The numbers given in parentheses are typical values for the constants on a 32-bit Unix system.

FLT_RADIX	radix of exponent rep	(2)
FLT_ROUNDS	floating point rounding mode	
FLT_DIG	decimal digits of precision	(6)
FLT_EPSILON	smallest x so $1.0\text{f} + x \neq 1.0\text{f}$	(1.1E − 7)
FLT_MANT_DIG	number of digits in mantissa	
FLT_MAX	maximum float number	(3.4E38)
FLT_MAX_EXP	maximum exponent	
FLT_MIN	minimum float number	(1.2E − 38)
FLT_MIN_EXP	minimum exponent	
DBL_DIG	decimal digits of precision	(15)
DBL_EPSILON	smallest x so $1.0 + x \neq 1.0$	(2.2E − 16)
DBL_MANT_DIG	number of digits in mantissa	
DBL_MAX	max double number	(1.8E308)
DBL_MAX_EXP	maximum exponent	
DBL_MIN	min double number	(2.2E − 308)
DBL_MIN_EXP	minimum exponent	

Send comments and corrections to J.H. Silverman, Math. Dept., Brown Univ., Providence, RI 02912 USA. (jhs@math.brown.edu)