Functions in C

Computing Lab
`https://www.isical.ac.in/~dfslab`

Indian Statistical Institute

Basics

- "Sub"-program
  e.g., `gcd`, `area of a triangle`
- Usually takes 1 or more *arguments* (input values)
- May compute a *return value* (output)
- Syntax

```
return_type function_name (argument1_type argument1,
            argument2_type argument2,
            ... )
{
     /* function body (local variables + statements) */
}
```

- `return` statement
  - `return;` or
  - `return expression;`
- `void`: special keyword to denote no arguments or return value

Using functions

Syntax

- In an expression:
  `x = ...  *  function_name(input1, input2, ...) -  ... ;`
- Only for the effect: `function_name(input1, input2, ...);`
- Ordering, number and types of inputs given to a function call *must match* with the ordering, number and types of arguments in the function header.

How does the computer handle function calls?

```
1   void main(void)
2   {  ...
3      m = f(x, y*z);
4      ...
5   }
6
7   int f(int a, int b)
8   {  ...
9      if (a > 0)
10        p = g(b);
11     else
12        p = h(b / 2);
13     return p;
14  }
15
16  int g(int m)
17  { ... }
18
19  int h(int n)
20  { ... }
```

1. Let `a = x`, `b = y*z`,
   i.e., copy `x`, `y*z` into `a`, `b`
2. Execute the statements in `f()`.
   (a) If `a` is positive, let `m = b`.
   Execute the statements in `g()`.
   (b) Otherwise, let `n = b/2`.
   Execute the statements in `h()`.
   (c) In either case, store the returned value in `p` (obtained from `g()` or `h()`).
   (d) Return the value `p` to the calling function (`main`).
   (e) Store the value returned by `f` in `u`.
3. Continue from line 4.

Prototypes

Prototype

```
return_type function_name (type1 argument1, type2 argument2, ... );
```

Example: `static double entropy(int, int);`

- Used to “announce” functions before they are actually written
- Argument types have to be specified, but argument names are optional
- Note the semicolon at the end
- Actual function may be written later within the same file, or in different file ⟵ LATER

Call by value

- A copy is made of any variable that is passed as an input argument to a function
- Changes to the variable inside the function are not visible outside
- **Exception:** *arrays* / *pointers*
  - used whenever a function needs to change an argument

Global, local variables

- *Global* variables: variables defined outside of the body of any function
  - stored in the data segment
- *Local* variables: variables defined within a function (or block)
  - not *visible* (cannot be used) outside the function
  - hides / masks any *global* variable with the same name
  - stored in a region of memory called an *activation record* or *stack frame*

Static variables

Defined within a function, but *not destroyed* when function returns
i.e., retains value across calls to the same function

Example:

```
void f(void) {
     static int i = 1;
     printf("This function has executed %d time(s)\n", i);
     i++;
}
```

Arrays / pointers and functions

Review question

For each of the following cases, what is the

1. type of `X`?
2. value of `sizeof(X)`?
3. value of `sizeof(X[0])`?

```
int *X[10];
```

```
int (*X)[10];
```

NOTE: `int X[][10]` generates a compiler error

Review question

For each of the following cases, what is the

1. type of `X`?
2. value of `sizeof(X)`?
3. value of `sizeof(X[0])`?

`int *X[10];` array of 10 pointers to integers

`sizeof(X)` : 80 bytes

`sizeof(X[0])` : 8 bytes

`int (*X)[10];`

NOTE: `int X[][10]` generates a compiler error

Review question

For each of the following cases, what is the

1. type of `X`?
2. value of `sizeof(X)`?
3. value of `sizeof(X[0])`?

`int *X[10];` array of 10 pointers to integers

`sizeof(X)` : 80 bytes

`sizeof(X[0])` : 8 bytes

`int (*X)[10];` pointer to an array of 10 integers

`sizeof(X)` : 8 bytes

`sizeof(X[0])` : 40 bytes

NOTE: `int X[][10]` generates a compiler error

One-dimensional array arguments – I

Equivalent (?) (see https://en.cppreference.com/w/c/language/array)

```
int f ( int A[100] , int size )
{
   ...
}
```

```
int f ( int A[] , int size )
{
   ...
}
```

"*When an array type is used in a function parameter list, it is transformed to the corresponding pointer type: int f(int a[2]) and int f(int* a) declare the same function. Since the function's actual parameter type is pointer type, a function call with an array argument performs array-to-pointer conversion; the size of the argument array is not available to the called function and must be passed explicitly.*"

But . . .

One-dimensional array arguments – II

See `array-args.c`

```
void print_array(int input[], int n);
void print_small_array(int input[LENGTH_SMALL], int n);
void print_large_array(int input[LENGTH_LARGE], int n);

int Small[LENGTH_SMALL], Large[LENGTH_LARGE], i;

print_array(Small, LENGTH_SMALL);
print_small_array(Small, LENGTH_SMALL);
print_large_array(Small, LENGTH_SMALL); // gives warning with -std=c11
print_array(Small, LENGTH_LARGE);
print_small_array(Small, LENGTH_LARGE);
print_large_array(Small, LENGTH_LARGE); // gives warning with -std=c11

/* Repeat with Large : no warnings */
```

Two- / multi-dimensional array args — I

```
void print_matrixA(int **matrix)
{
    int i, j;
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            printf("%d ", matrix[i][j]);
        }
        putchar('\n');
    }
}

void print_matrixB(int (*matrix)[COLS]) // try changing COLS to ROWS
{
    int i, j;
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            printf("%d ", matrix[i][j]);
        }
        putchar('\n');
```

Two- / multi-dimensional array args — II

```
void print_matrixC(int matrix[ROWS][COLS]) // try changing COLS to ROWS
{
    int i, j;
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            printf("%d ", matrix[i][j]);
        }
        putchar('\n');
    }
}

void print_matrixD(int **matrix)
{
    int i, *base = (int *) matrix;
    for (i = 0; i < ROWS*COLS; i++) {
        if (i % COLS == 0)
            putchar('\n');
        printf("%d ", base[i]);
```

Pointer-to-pointer vs. 2-dimensional arrays

```c
int main()
{
    /* IMPORTANT NOTE about initialisation: */
    int a[ROWS][COLS] = {0};
    int i, j, *p, **pp;

    for (i=0; i<ROWS; i++)
        for (j=0; j<COLS; j++)
            a[i][j] = i*COLS + j;
    pp = (int **) a;
    printf("%p %lu\n", pp, pp);

    p = (int *) a;
    printf("Start: %p %lu\nEnd: %p %lu\n",  // ignore warnings about %lu
           p, p, p+ROWS*COLS, p+ROWS*COLS);

    p += ROWS;
    printf("%d\n", *p);
    p += (COLS - ROWS);
    printf("%d\n", *p);
    p = (int *) a + ROWS*COLS - 1;
    printf("%d\n", *p);
```

a

0	1	2
3	4	5

Layout of a in memory

0	1	2	3	4	5

Pointer-to-pointer vs. 2-dimensional arrays

```c
int main()
{
    /* IMPORTANT NOTE about initialisation: */
    int a[ROWS][COLS] = {0};
    int i, j, *p, **pp;

    for (i=0; i<ROWS; i++)
        for (j=0; j<COLS; j++)
            a[i][j] = i*COLS + j;
    pp = (int **) a;
    printf("%p %lu\n", pp, pp);

    p = (int *) a;
    printf("Start: %p %lu\nEnd: %p %lu\n",  // ignore warnings about %lu
           p, p, p+ROWS*COLS, p+ROWS*COLS);

    p += ROWS;
    printf("%d\n", *p);
    p += (COLS - ROWS);
    printf("%d\n", *p);
    p = (int *) a + ROWS*COLS - 1;
    printf("%d\n", *p);
```

```
0x7ffc17090d00 140720694955264
```

```
Start:  0x7ffc17090d00 140720694955264
End:  0x7ffc17090d18 140720694955288
```

a

0	1	2
3	4	5

Layout of a in memory

0	1	2	3	4	5

Pointer-to-pointer vs. 2-dimensional arrays (contd.)

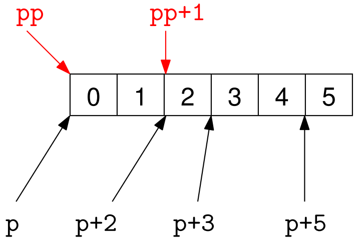
```
puts("With print_matrixB:");
print_matrixB(a);
/* print_matrixB(pp); // ERROR: expected int (*)[3], passed int **
*/

puts("With print_matrixC:");
print_matrixC(a);
/* print_matrixC(pp); // ERROR: expected int (*)[3], passed int **
*/

printf("With print_matrixD:");
/* print_matrixD(a);  // ERROR: expected int **, passed int (*)[3]
*/
print_matrixD(pp);

puts("\nWith print_matrixA:");
/* print_matrixA(a);  // ERROR: expected int **, passed int (*)[3]
*/
print_matrixA(pp);    // Segmentation fault
```

Pointer-to-pointer vs. 2-dimensional arrays (contd.)



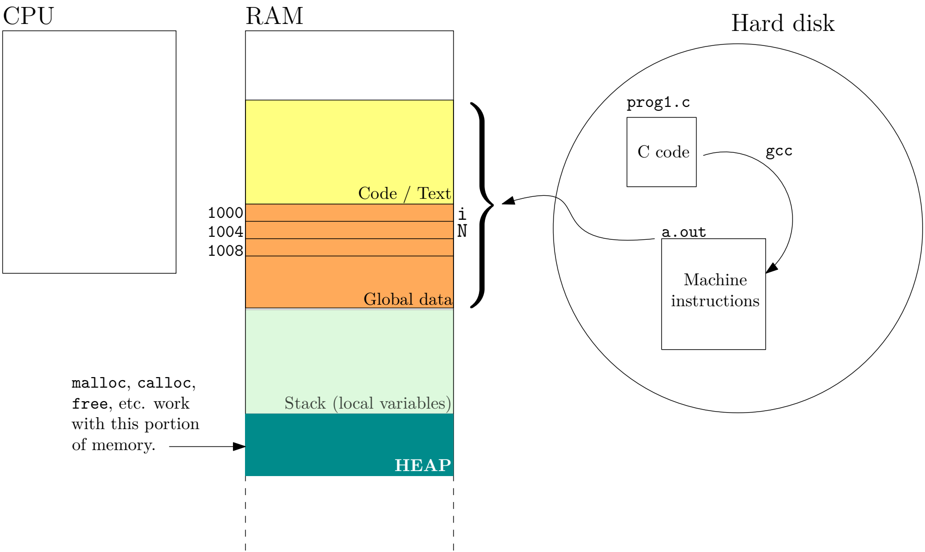
1. What is the *type* of `pp+1`?
2. What is the *type* of `pp[1]`?
3. What is the *value* of `pp[1]`?
4. What is the *location* of `pp[1][0]`, `pp[1][1]`, ... ?

Lessons learnt

- Different compilers / different versions of the same compiler are based on different language standards
- Safest to write code that will compile and work across multiple versions
- When passing matrices / multi-dimensional arrays as function arguments, safest + most flexible to
  - use `malloc`, `calloc`, etc. to dynamically allocate storage (use `matrix_alloc` and `matrix_free` from `common.h` if necessary)
  - pass pointers of the form `ELEMENT_TYPE **` along with dimensions

Digression regarding memory / storage

Simplified view of a program's memory



Activation records

Where are the activation records (AR) stored?

- Simple solution: AR == one fixed block of memory per function
  LATER: does not work for recursive functions
- Better solution: one block of memory per *function call*
  - AR allocated / deallocated when function is called / returns
  - variables created when function is called; destroyed when function returns
  - need to keep track of *nested* calls
  - function calls behave in *last in first out* manner
    $\Rightarrow$ use *stack* to keep track of ARs

Activation stack

- Activation records stored in a chunk of memory called *activation stack*
- When a function is called, its activation record is added to the end of the activation stack.
- When function returns, its activation record is removed.
- LATER: works for recursive functions

Activation stack: example

```c
void main(void)
{   ...
    m = f(x, y*z);
    ...
}

int f(int a, int b)
{   ...
    if (a > 0)
       p = g(b);
    else
       p = h(b / 2);
    return p;
}

int g(int m)
{ ... }

int h(int n)
{ ... printf(...); ...}
```

Practice problems I

1. Implement a list of integers using dynamic memory. The list consists of a header storing two items:
   - the size of the list, and
   - a dynamically allocated array capable of storing **only** the elements in the list.

   Define a structure to store the size of the list, and a dynamically allocated array that stores the list elements. Use a `typedef` to define `LIST` as the name of your structure.

   Implement the following functions.

   (a) `create_list(void)`: returns an empty list. For an empty list, the size is zero, and the array is NULL.

   (b) `print_list(LIST L)`: print the elements of the list separated by spaces, and terminated by a newline.

Practice problems II

(c) `append(LIST L, int a)`: appends `a` to the end of the list `L`, and returns the modified list.

(d) `prepend(LIST L, int a)`: prepends `a` at the beginning of the list `L`, and returns the modified list.

(e) `deletelast(LIST L)`: deletes the last element of the list, and returns the modified list.

(f) `deletefirst(LIST L)`: deletes the first element of the list, and returns the modified list.

(g) `deleteall(LIST L, int a)`: deletes all occurrences of `a` in `L`, and returns the modified list.

Do not forget to use `free()` where necessary.

Practice problems III

Test your code using, for example, the following sequence of operations.

1	2	3	4
create_list()	append(2)	prepend(8)	deletefirst()
append(9)	append(3)	prepend(9)	deletefirst()
append(1)	prepend(9)	prepend(1)	deletefirst()
append(2)	prepend(2)	deletelast()	deletefirst()
append(3)	prepend(4)	deletelast()	deleteall(7)
append(6)	prepend(8)	deletelast()	deleteall(2)
append(7)	prepend(2)	deletelast()	deleteall(9)
append(8)	prepend(5)	deletelast()	deleteall(4)
append(4)	prepend(9)	deletefirst()	deleteall(7)

SOURCE: IIT Kharagpur, CS19001/CS19002 Programming and Data Structures Laboratory, 2015, Assignment 8.

Practice problems IV

2. Let `s` and `t` be strings. Implement the following functions in C:
   (a) `strlen(s)`: returns the length of `s`, i.e., the number of characters present in `s`;
   (b) `strcmp(s, t)`: returns 1 if `s` and `t` are identical, 0 otherwise;
   (c) `diffByOne(s, t)`: returns 1 if `s` and `t` are of the same length, and differ in exactly one position, 0 otherwise.

      Thus, if `s` is `sale` and `t` is either `salt` or `same` or `pale`, `diffByOne(s, t)` should return 1; but if `s` and `t` are `salt` and `salty`, it should return 0.
3. Write a function `uniquify` that
   - takes two arguments: an array `A` containing non-negative integers, and `n`, the number of integers contained in the array, and
   - stores in `A` a list of the *distinct* integers contained in `A` if the integers in `A` are sorted in increasing order, and returns the number of distinct integers in `A`; and returns -1 otherwise.

Practice problems V

4. Recall that the `rand()` function provided by the standard C library generates a sequence of pseudo-random integers uniformly sampled from the range $[0, \text{RAND\_MAX}]$. For the first part of this question, you have to use the *Box-Muller transform* (described below) to generate a pseudo-random sequence drawn from a normal distribution. [5]
   Suppose u_1 and u_2 are independent samples chosen from the uniform distribution on the unit interval $(0, 1)$. Let

$$
\begin{aligned}
z_0 &= \sqrt{-2 \ln u_1} \cos(2\pi u_2) \\
z_1 &= \sqrt{-2 \ln u_1} \sin(2\pi u_2)
\end{aligned}
$$

   Then z_0 and z_1 are independent random variables with a standard normal distribution.

Write a function that uses the above idea to generate a pseudo-random sequence drawn from a standard normal distribution. Your function should have the following prototype:

```
double normal(void);
```

5. A person's daily commute from home to office consists of three stages: she first takes an auto to the nearest Metro station A, then takes the Metro from station A to station B, and finally takes a bus from B to her office. Assume that the time taken to complete each stage follows an independent normal distribution $N(\mu_i, \sigma_i^2)$ for $i = 1, 2, 3$. You have to write a program that uses simulation to estimate the probability that the person's commute will take more than a specified amount of time T.

   Input format. The values of $\mu_1, \sigma_1, \mu_2, \sigma_2, \mu_3, \sigma_3$ and T (all in minutes) will be provided in that order via standard input.

Practice problems VII

Output format. Your program should print a single floating point number corresponding to the desired probability, correct to 8 decimal places.

Method. You should use the function that you wrote above to simulate 10,000 journeys, and report the proportion of these journeys that took more than T minutes.

NOTE: If a random variable X follows the standard normal distribution, then the random variable $Y = aX + b$ (where a, b are real constants) follows a normal distribution with mean b and variance a^2.