C language reminder

Part I. Reminder of the C programming language

Frédéric S. Masset

Instituto de Ciencias Físicas, UNAM

Outline

Outline

1. Basic structure of a C program
2. Pointers and allocations

Basics of C

A typical C program looks like this

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

int main ()
{
   printf ("This is a random number: %f\n", drand48());
   return 0;
}
```

example1.c

This C program can be compiled with: `cc example1.c`
This should produce an executable file named `a.out`, which can be run with: `./a.out`

Basics of C

A typical C program looks like this

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

int main ()
{
   printf ("This is a random number: %f\n", drand48());
   return 0;
}
```

Libraries needed by the program

Basics of C

A typical C program looks like this

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

int main ()
{
   printf ("This is a random number: %f\n", drand48());
   return 0;
}
```

`stdio.h` (standard input/output) provides `printf ()`

Basics of C

A typical C program looks like this

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

int main ()
{
   printf ("This is a random number: %f\n", drand48());
   return 0;
}
```

stdlib.h (standard library) provides drand48 ()

Basics of C

A typical C program looks like this

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

int main ()
{
   printf ("This is a random number: %f\n", drand48());
   return 0;
}
```

Instructions are terminated by a ; symbol. Usually one instruction per line

Basics of C

A typical C program looks like this

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

int main ()
{
   printf ("This is a random number: %f\n", drand48());
   return 0;
}
```

Any C program must contain a `main ()` function, which is where the execution flow starts upon execution

Basics of C

A typical C program looks like this

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

int main ()
{
   printf ("This is a random number: %f\n", drand48());
   return 0;
}
```

The beginning and end of a function are marked with brackets {. . . }. The same is true for the beginning/end of loops, etc. There are like the `begin` and `end` of FORTRAN

Basics of C

A typical C program looks like this

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

int main ()
{
   printf ("This is a random number: %f\n", drand48());
   return 0;
}
```

All C functions must be followed by ()'s, **even if the list of arguments is empty**

Basics of C

A typical C program looks like this

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

int main ()
{
   printf ("This is a random number: %f\n", drand48());
   return 0;
}
```

The `main` function returns an integer (int), which is here 0.

Basics of C

A typical C program looks like this

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

int main ()
{
   printf ("This is a random number: %f\n", drand48());
   return 0;
}
```

The `printf ()` function prints its argument on the terminal. The `%f` symbol is substituted by the floating point value found after the string (here the random number returned by `drand48 ()`).

Basics of C

A typical C program looks like this

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

int main ()
{
   printf ("This is a random number: %f\n", drand48());
   return 0;
}
```

The symbol `\n` indicates a carriage return

Writing a simple C function

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

float rd () {
   return (drand48()-0.5)*2.0;
}

int main () {
   int i;
   for (i = 0; i < 5; i++) {
      printf ("Random number #%d : %f\n", i, rd());
   }
   return 0;
}
```

This is `example2.c`

Writing a simple C function

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

float rd () {
   return (drand48()-0.5)*2.0;
}

int main () {
   int i;
   for (i = 0; i < 5; i++) {
      printf ("Random number #%d : %f\n", i, rd());
   }
   return 0;
}
```

This is our function, called `rd ()` (like *random*)

Writing a simple C function

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

float rd () {
   return (drand48()-0.5)*2.0;
}

int main () {
   int i;
   for (i = 0; i < 5; i++) {
      printf ("Random number #%d : %f\n", i, rd());
   }
   return 0;
}
```

It returns a float random number uniformly distributed between -1 and $+1$

Writing a simple C function

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

float rd () {
   return (drand48()-0.5)*2.0;
}

int main () {
   int i;
   for (i = 0; i < 5; i++) {
      printf ("Random number #%d : %f\n", i, rd());
   }
   return 0;
}
```

We have inserted a loop on the integer `i`

Writing a simple C function

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

float rd () {
   return (drand48()-0.5)*2.0;
}

int main () {
   int i;
   for (i = 0; i < 5; i++) {
      printf ("Random number #%d : %f\n", i, rd());
   }
   return 0;
}
```

We have inserted a loop on the integer `i`

Writing a simple C function

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

float rd () {
   return (drand48()-0.5)*2.0;
}

int main () {
   int i;
   for (i = 0; i < 5; i++) {
      printf ("Random number #%d : %f\n", i, rd());
   }
   return 0;
}
```

In C we must declare **all** variables

Writing a simple C function

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

float rd () {
   return (drand48()-0.5)*2.0;
}

int main () {
   int i;
   for (i = 0; i < 5; i++) {
      printf ("Random number #%d : %f\n", i, rd());
   }
   return 0;
}
```

`i++` means `i = i+1`. `i` is incremented after each execution of the loop.

Writing a simple C function

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

float rd () {
   return (drand48()-0.5)*2.0;
}

int main () {
   int i;
   for (i = 0; i < 5; i++) {
      printf ("Random number #%d : %f\n", i, rd());
   }
   return 0;
}
```

The loop begins with `i = 0`

Writing a simple C function

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

float rd () {
   return (drand48()-0.5)*2.0;
}

int main () {
   int i;
   for (i = 0; i < 5; i++) {
      printf ("Random number #%d : %f\n", i, rd());
   }
   return 0;
}
```

The loop is executed while `i < 5` is true

Writing a simple C function

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

float rd () {
   return (drand48()-0.5)*2.0;
}

int main () {
   int i;
   for (i = 0; i < 5; i++) {
      printf ("Random number #%d : %f\n", i, rd());
   }
   return 0;
}
```

The loop is therefore executed 5 times, for `i=0,1,2,3,4`

Writing a simple C function

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

float rd () {
   return (drand48()-0.5)*2.0;
}

int main () {
   int i;
   for (i = 0; i < 5; i++) {
      printf ("Random number #%d : %f\n", i, rd());
   }
   return 0;
}
```

The `%d` symbol is substituted by the first (integer) value found after the string (here the variable `i`)

Writing a simple C function

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

float rd () {
   return (drand48()-0.5)*2.0;
}

int main () {
   int i;
   for (i = 0; i < 5; i++) {
      printf ("Random number #%d : %f\n", i, rd());
   }
   return 0;
}
```

The %f symbol is substituted by the second (floating) value found after the string (here the value returned by rd ())

Manual pages

Any doubt on a C function ? Check the manual pages.
E.g., try:

```
man drand48
```

and you will get a lot of information about the generation of random values in C, as well as the name of the library that provides the function (here `stdlib.h`), and comprehensive information about how to invoke the function.

Pointers : example3.c

```
#include <stdio.h>

void main () {
   float b = 3.0;
   float *ptr;
   ptr = &b;
   printf ("ptr  = %ld\n", ptr);
   printf ("*ptr = %f\n", *ptr);
   *ptr = 4.0;
   printf ("b    = %f\n", b);
}
```

Pointers : example3.c

```
#include <stdio.h>

void main () {
   float b = 3.0; Declare and initialize
   float *ptr;
   ptr = &b;
   printf ("ptr  = %ld\n", ptr);
   printf ("*ptr = %f\n", *ptr);
   *ptr = 4.0;
   printf ("b    = %f\n", b);
}
```

Pointers : example3.c

```
#include <stdio.h>

void main () {
   float b = 3.0;
   float *ptr; ptr is an address of a float
   ptr = &b;
   printf ("ptr  = %ld\n", ptr);
   printf ("*ptr = %f\n", *ptr);
   *ptr = 4.0;
   printf ("b    = %f\n", b);
}
```

Pointers : example3.c

```
#include <stdio.h>

void main () {
   float b = 3.0;
   float *ptr;
   ptr = &b; &b is the address of b
   printf ("ptr  = %ld\n", ptr);
   printf ("*ptr = %f\n", *ptr);
   *ptr = 4.0;
   printf ("b    = %f\n", b);
}
```

Pointers : example3.c

```
#include <stdio.h>

void main () {
   float b = 3.0;
   float *ptr;
   ptr = &b; &b is the address of b, stored in ptr
   printf ("ptr  = %ld\n", ptr);
   printf ("*ptr = %f\n", *ptr);
   *ptr = 4.0;
   printf ("b    = %f\n", b);
}
```

Pointers : example3.c

```
#include <stdio.h>

void main () {
   float b = 3.0;
   float *ptr;
   ptr = &b;
   printf ("ptr  = %ld\n", ptr); long ugly integer
   printf ("*ptr = %f\n", *ptr);
   *ptr = 4.0;
   printf ("b    = %f\n", b);
}
```

Pointers : example3.c

```
#include <stdio.h>

void main () {
   float b = 3.0;
   float *ptr;
   ptr = &b;
   printf ("ptr  = %ld\n", ptr);
   printf ("*ptr = %f\n", *ptr); value of float found there
   *ptr = 4.0;
   printf ("b    = %f\n", b);
}
```

Pointers : example3.c

```
#include <stdio.h>

void main () {
   float b = 3.0;
   float *ptr;
   ptr = &b;
   printf ("ptr  = %ld\n", ptr);
   printf ("*ptr = %f\n", *ptr);
   *ptr = 4.0; Change the value at address ptr
   printf ("b    = %f\n", b);
}
```

Pointers : example3.c

```
#include <stdio.h>

void main () {
   float b = 3.0;
   float *ptr;
   ptr = &b;
   printf ("ptr  = %ld\n", ptr);
   printf ("*ptr = %f\n", *ptr);
   *ptr = 4.0;
   printf ("b    = %f\n", b); amounts to changing b !
}
```

Array allocation : example4.c

```
#define N 10

void main () {
   float *a, *b;
   int i;
   a = malloc (N*sizeof(float));
   b = malloc (N*sizeof(float));
   for (i = 0; i < N; i++) {
      a[i] = drand48 ();
      b[i] = drand48 ();
      printf ("a[%d] = %f and b[%d] = %f\n",i,a[i],i,b[i]);
   }
   free (a);
   free (b);
}
```

Array allocation : example4.c

```
#define N 10 Preprocessor instruction (begins with a #)

void main () {
   float *a, *b;
   int i;
   a = malloc (N*sizeof(float));
   b = malloc (N*sizeof(float));
   for (i = 0; i < N; i++) {
      a[i] = drand48 ();
      b[i] = drand48 ();
      printf ("a[%d] = %f and b[%d] = %f\n",i,a[i],i,b[i]);
   }
   free (a);
   free (b);
}
```

Array allocation : example4.c

```
#define N 10

void main () {
   float *a, *b; a and b are pointers to floats
   int i;
   a = malloc (N*sizeof(float));
   b = malloc (N*sizeof(float));
   for (i = 0; i < N; i++) {
      a[i] = drand48 ();
      b[i] = drand48 ();
      printf ("a[%d] = %f and b[%d] = %f\n",i,a[i],i,b[i]);
   }
   free (a);
   free (b);
}
```

Array allocation : example4.c

```
#define N 10

void main () {
   float *a, *b; They are not initialized
   int i;
   a = malloc (N*sizeof(float));
   b = malloc (N*sizeof(float));
   for (i = 0; i < N; i++) {
      a[i] = drand48 ();
      b[i] = drand48 ();
      printf ("a[%d] = %f and b[%d] = %f\n",i,a[i],i,b[i]);
   }
   free (a);
   free (b);
}
```

Array allocation : example4.c

```
#define N 10

void main () {
   float *a, *b;
   int i;
   a = malloc (N*sizeof(float)); sizeof(float)=4
   b = malloc (N*sizeof(float));
   for (i = 0; i < N; i++) {
      a[i] = drand48 ();
      b[i] = drand48 ();
      printf ("a[%d] = %f and b[%d] = %f\n",i,a[i],i,b[i]);
   }
   free (a);
   free (b);
}
```

Array allocation : example4.c

```
#define N 10

void main () {
   float *a, *b;
   int i;
   a = malloc (N*sizeof(float));
   b = malloc (N*sizeof(float));
   for (i = 0; i < N; i++) {
      a[i] = drand48 ();
      b[i] = drand48 ();
      printf ("a[%d] = %f and b[%d] = %f\n",i,a[i],i,b[i]);
   }
   free (a);
   free (b);
}
```

malloc (n) reserves n bytes and returns the start address

Array allocation : example4.c

```
#define N 10

void main () {
   float *a, *b;
   int i;
   a = malloc (N*sizeof(float));
   b = malloc (N*sizeof(float));
   for (i = 0; i < N; i++) {
      a[i] = drand48 (); We initialize the arrays
      b[i] = drand48 (); a and b with random values
      printf ("a[%d] = %f and b[%d] = %f\n",i,a[i],i,b[i]);
   }
   free (a);
   free (b);
}
```

Array allocation : example4.c

```
#define N 10

void main () {
   float *a, *b;
   int i;
   a = malloc (N*sizeof(float));
   b = malloc (N*sizeof(float));
   for (i = 0; i < N; i++) {
      a[i] = drand48 ();
      b[i] = drand48 ();
      printf ("a[%d] = %f and b[%d] = %f\n",i,a[i],i,b[i]);
   }
   free (a); We deallocate a and b before leaving
   free (b); Otherwise : memory leak !
}
```

Modifying a variable with a function

example5.c

We write a little function supposed to change the value of its argument.

```c
#include <stdio.h>

void myfunc (float a) {
  a = a * 2.0;
}

int main () {
  float b = 23.0;
  myfunc (b);
  printf ("%f\n", b);
  return 0;
}
```

Modifying a variable with a function

example5.c

We write a little function supposed to change the value of its argument.

```
#include <stdio.h>

void myfunc (float a) {
  a = a * 2.0;
}

int main () {
  float b = 23.0;
  myfunc (b);
  printf ("%f\n", b);  23.0000 !!
  return 0;            we expected 46.000 !!
}
```

Modifying a variable with a function

Arguments are passed by value, not by reference (they are copied to the stack, where they are read by the calling function). Any modification of the value within the function is **not** recognized outside of its scope ⇒ we need to cheat !

This is example6.c

```
#include <stdio.h>

void myfunc (float *a) {
  *a = (*a) * 2.0;
}

int main () {
  float b = 23.0;
  myfunc (&b);
  printf ("%f\n", b);
  return 0;
}
```

Modifying a variable with a function

Arguments are passed by value, not by reference (they are copied to the stack, where they are read by the calling function). Any modification of the value within the function is **not** recognized outside of its scope ⇒ we need to cheat !

```
#include <stdio.h>     This is example6.c

void myfunc (float *a) {
  *a = (*a) * 2.0;
}

int main () {
  float b = 23.0;
  myfunc (&b);  passed by reference
  printf ("%f\n", b);  46.0000 !!
  return 0;            this time it works
}
```

Modifying a variable with a function

- We cannot modify directly the argument `foo` of a function in C.
- The only way is to pass it by reference, *i.e.* to give its address (`&foo`) rather than its value itself. In C, this workaround is unavoidable.

About pointer's syntax

The syntax used to declare and invoke pointers can be confusing. However, it is extremely logical. You should think of `anytype *` as meaning “a pointer to a variable of type” `anytype`.

Hence if you see `float *a`, you can think of it in two different ways:

- `a` is a pointer to a float
- `*a` is a float
- Both conceptions are equivalent and valid.

About pointer's syntax

The syntax used to declare and invoke pointers can be confusing. However, it is extremely logical. You should think of `anytype *` as meaning “a pointer to a variable of type” `anytype`.

Hence if you see `float *a`, you can think of it in two different ways:

- `a` is a pointer to a float
- `*a` is a float
- Both conceptions are equivalent and valid.

About pointer's syntax

The syntax used to declare and invoke pointers can be confusing. However, it is extremely logical. You should think of `anytype *` as meaning “a pointer to a variable of type” `anytype`.

Hence if you see `float *a`, you can think of it in two different ways:

- `a` is a pointer to a float
- `*a` is a float
- Both conceptions are equivalent and valid.

About dynamic allocation

- The `malloc ()` instruction allows to reserve memory at runtime (amount not known in advance). Very useful in C, widely used.
- Nothing prevents the programmer from doing an out-of-bound memory write or read. This can lead to severe errors (*e.g.* segmentation fault) which may be hard to debug.

About dynamic allocation

- The `malloc ()` instruction allows to reserve memory at runtime (amount not known in advance). Very useful in C, widely used.
- Nothing prevents the programmer from doing an out-of-bound memory write or read. This can lead to severe errors (*e.g.* segmentation fault) which may be hard to debug.

Interaction with user

Informing the user

Print information to terminal (standard output) using: `printf ()` or `fprintf (stdout, ...)`

Getting information from user

Data can be input using: `fscanf ()`. For instance, assume we read from keyboard a floating value into the variable `foo`. How do we do that ? `fscanf (stdin, “%f", foo)` or `fscanf (stdin, "%f", &foo)` ? Why ?

Exercise 1

Write a simple integrator (Riemann sum), filling the following skeleton:

```
 Include adequate libraries

 float myfunc (float x) {
   Your choice here for a function f(x)
 }

 int main () {
   float a, b; // Limits of the integration range
   int i;
   float t;

   printf (“Please input a = “);
   fscanf (stdin, “%f", up to you...);
   printf (“Please input b = “);
   fscanf (stdin, “%f", up to you...);

   Write a loop on i to evaluate the integral of f from a to b

   Print the result
}
```