COMP26120: Algorithms and Imperative Programming

Lecture 5:
Program structuring, Java vs. C,
and common mistakes

Lecture outline

- Program structuring
  - Functions (defining a functions, passing arguments and parameters, return values, call by reference vs. call by value, program parameters - argc,argv);
- Java classes, etc. vs. C
  - Mechanisms in C that can "make up" for not having classes;
  - Program partitioning and compilation;
  - Identifier visibility;
- Common mistakes
- General remark: no lecture next week, but you will have a lab; make sure that you get as many of your first semester work marked!

Program structuring
Structured programming

- Disciplined approach to programming that (hopefully) produces easy to understand, debug, and maintain programs.
- Common advice:
  1. Use sensible and meaningful variable names.
  2. Use comments to describe the program functionality.
  3. Do not use obscure programming tricks.
- Recall implicit type conversion we discussed earlier in this context – this may cause problems during the execution!

Program structuring
Modularity

- If top-down design of computer programs is adopted, the task is subdivided into a number of smaller connected tasks up to a point when these can be implemented in isolation.
- The relationships between sub-tasks can be represented by structure diagrams, that can be refined recursively.
- Elementary sub-tasks can be programmed by procedures (called functions in C).
- Functions may have a number of input parameters, and return values (results).

Program structuring
Definition of a function

```
int fun1 (int a, float b);
float localvar;
int outvar;
{
…
  outvar = …
  return outvar;
}
```

- Function type – defines the type of the return value (if no value is returned the type is void);
- The function name is a unique identifier (the same rules apply as for the variable names.
- A parameter list of the input arguments that needs to be passed to the function.
- Definition(s) of the internal (local) variable(s).
- Output (return) variable.
- Function body.

Program structuring
Functions

- C standard does not support function nesting. However, gcc compiler has this feature (should not be used if code portability is an issue.
- Example.

```
int f1(void)
{ …
  int f2(int x)
  {…
     return x;
  }
…
return 0;
}
```

```
int f2(int x)
{…
   return x;
}
int f1(void)
{…
    f2(x);
 …
 return 0;
}
```

Program structuring
Functions

- A function may return a value of certain type (specified by the type of the function) using the `return` statement. The calling program does not have to do anything with this value. If the function is defined as `void`, it returns no value. If `void` is present in the argument list, the function takes no input arguments.
- Example. `void my_fun(void)`

Program structuring
Functions

- Further notes about functions:
  1. The compiler checks function calls against the definitions.
  2. The number of parameters and their types in the call must match the number and type in the function definition.
  3. If the result type is not specified in the function definition, the compiler assumes it will return int, but always double-check (it may be compiler dependent).
  4. If you do not specify a parameter type, it is assumed to be int, but always double-check.
  5. Do not put a semi-colon after the parameter list.

Program structuring
Calling functions by value

- Example.

```
void change_letter(char a)
{
    a = “S”;
}
int main (void)
{
    char a = “P”;
    change_letter(a);
    printf(“The character is: %c \n”, a);
    return 1;
}
```

- Question: What is the value of a after the execution of the code? Why ‘P’?

Program structuring
Calling functions by reference

- If we want to manipulate the actual variable, rather than its copy, it has to be passed by reference. In some programming languages this can be done directly (Java?), but in C it has to be simulated.
- How to do this? Remember the operators & (address of) and * (dereference)?
- Example. Write a modified function change_letter which will produce the answer 'S'.

```
void change_letter(*char a)
{
   *a = "S";
}
```

```
change_letter(&a);
```

Program structuring
Calling functions by reference

- This is a simulated call by reference. In this context the address (or pointer) of the character variable **a** is passed by value.
- Since the function **change_letter** manipulates directly the variables pointed by the function arguments, it looks as the variables are passed by reference.

Program structuring
Arrays as function parameters

- Example: Write a function that echoes an array of characters which ends with \0 to the screen (recall ARM assembler).

```
void pstring(char message[ ])
 {
     int i = 0;
     while(message[i] != '\0')
    {
        printf("%c", message[i]); i++;
    }
 }
```

- The function accepts an array as an argument, which dimension we do not need to specify. Why? The arrays are passed to functions by reference, i.e. The function receives the pointer to the first element of the array.

Program structuring
Arrays as function parameters

- The function **pstring** is called from the main program as:

```
int main (void)
 {
      char name[ ] = "Peter\0";
      pstring(name);
      return 0;
 }
```

- Question: What will be the value of **name** if the first line of the function **pstring** is changed to

  message[ ]="Steven\0";

- Remember, arrays are passed by reference, so **message** and **name** point to the same location.

Program structuring
Structures as function parameters

- How to pass a structure to a function?

```
struct person
{
    char name[20];
    char address[32];
    unsigned int age;
    float height;
};
```

```
void print_human (struct person tutor)
{
    printf("Name: %s\n Address: %s\n
            Age: %d\n Height: %6.2f",
            person.name, person.address,
            person.age, person.height);
}
```

- This passes the structure by value. What is wrong with it?
- The code will work, but it will be inefficient. Why? Passing by value makes a copy of the object – what happens if the structure is large, and you need to print it a million times?

Program structuring
Structures as function parameters

- What is the (obvious) solution? Pass a pointer to the structure that needs to be printed:

```
void print_human (struct person tutor)
{
   printf("Name: %s\n Address: %s\n
           Age: %d\n Height: %6.2f",
           person.name, person.address,
           person.age, person.height);
}
```

```
void print_human (struct person *tutor)
{
    printf("Name: %s\n Address: %s\n
           Age: %d\n Height: %6.2f",
           person->name, person->address,
           person-> age, person ->height);
}
```

Program structuring
Returning structures from functions

- How to return a structure from a function? As in the previous case, passing it by value is inefficient for the same reasons.

```
struct person *add_person (char *name, char *address,
                           unsigned int age, float height)
{
    struct person *new_person = NULL;
    new_person = (struct person*)malloc(sizeof(struct person));
    if(new_person != NULL) /* Will be discusses under "Mistakes"*/
    {
        strcpy(new_person -> name, name);
        strcpy(new_person -> address, address);
        new_person -> age = age;
        newhuman -> height = height;
    }
    return new_person;
}
```

Program structuring
Program parameters

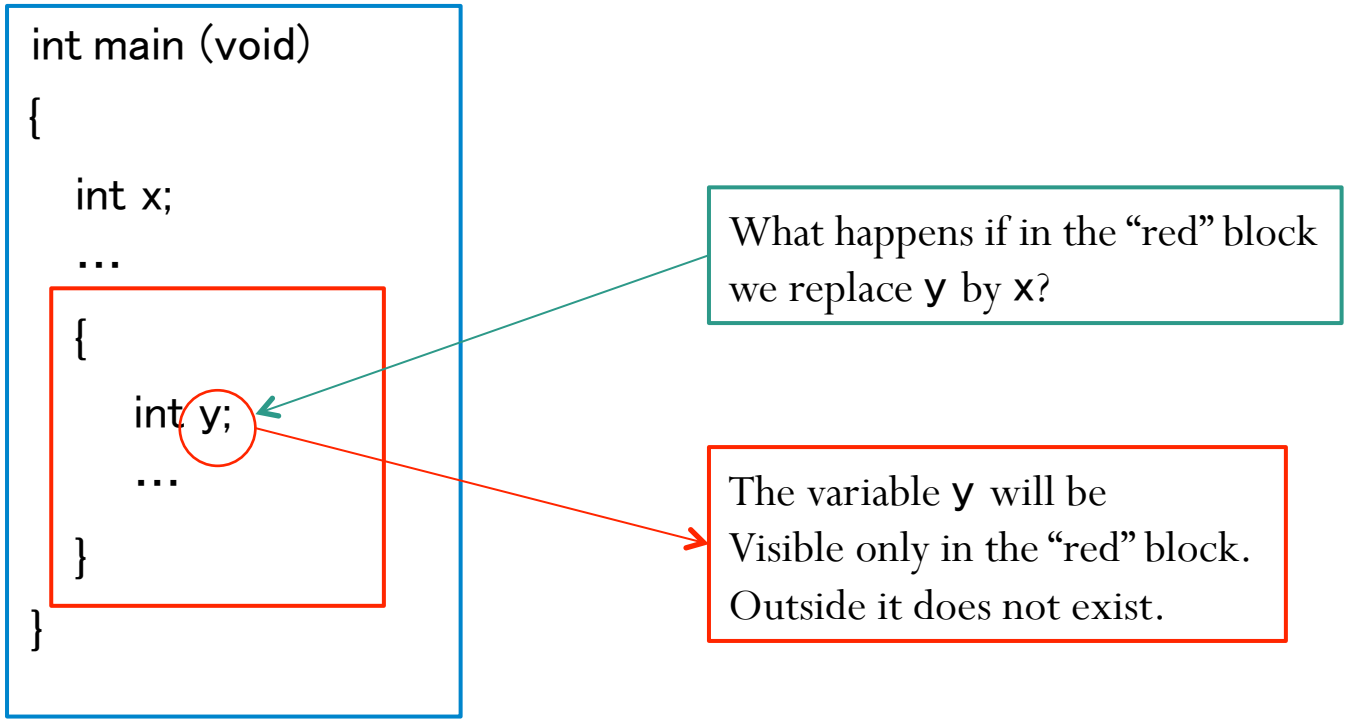
- The main program is a function called by the OS. Why is `main` defined to be `int`?
- Returning 0 from `main` indicates successful completion.
- Input parameters can be passed to the program via the command line using two parameters: `argc` and `argv`.
- `argc` is an integer giving a number of command line arguments (including the program name!).
- `argv` is an array of character pointers to command line arguments.

```
add_tutor_height  182.0
```

- You need to convert the string "182.0" into float.

Program partitioning
Blocks and scoping

- The code enclosed inside a pair of curly brackets is called a block.
- An important feature of blocks is that they can have their local variables.



Program partitioning
Variable definition and declaration

- What is the difference between variable definition and declaration?
- Variable declaration states the properties of the variable (its name and type), but does not create the object (allocate memory). Its purpose is to notify the compiler that such variable will be used at some point and to check for inconsistencies.
- Variable definition allocates storage for the object in memory and creates the object.

Variable declaration	Variable definition
`extern int my_int;`	`int my_int;`

Program partitioning
Function prototypes

- Function prototypes are an analogue to variable declarations.

```
float my_function(float f, int i);
```

- Why is this not a function definition? Something is missing?
- Why do you need function prototypes?

```
float my_function(float f, int i);
...
int my_function (float f, int i)
{ float answer; answer = f * i; return answer; }
```

Program partitioning
Storage classes

- Each variable in a program has its type and its storage class.
- C has five storage classes (auto, extern, static, register, and const).
- Variables defined inside the program blocks are auto variables. They are created on a stack upon entering a block and destroyed when the block is exited. Using the keyword auto is redundant.
- Variables defined outside the functions (including main) are visible by all functions. They are global variables and are of the storage class extern.
- Variables defined inside program blocks or functions can be made permanent (but accessible only within this block/function) by using the storage class static. What is the Java equivalent of this? (private)
- If a local variable is used often, it can be declared as register, hinting to the compiler that it should be kept in a processor register.
- If a variable value is not expected to change during the program execution, it should be declared as const.

Program partitioning
Storage classes

one.java

```
class One
{
   public static int x;
   private static int y;
   private static void a (int b)
   {
      y= y − b; x= x + b; Two.z= b; Two.c (b);
   }
}
```

one.h

```
extern int x;
```

one.c

```
#include ″one.h″
#include ″two.h″
int x;
static int y;
static void a (int b)
{    y= y − b; x= x + b; z= b; c (b);    }
```

two.java

```
class Two
{
   public static int z;
   private static int w;
   public static void c (int d)
   {
      z= w − d; One.x= d;
   }
}
```

two.h

```
extern int z; extern void c (int d);
```

two.c

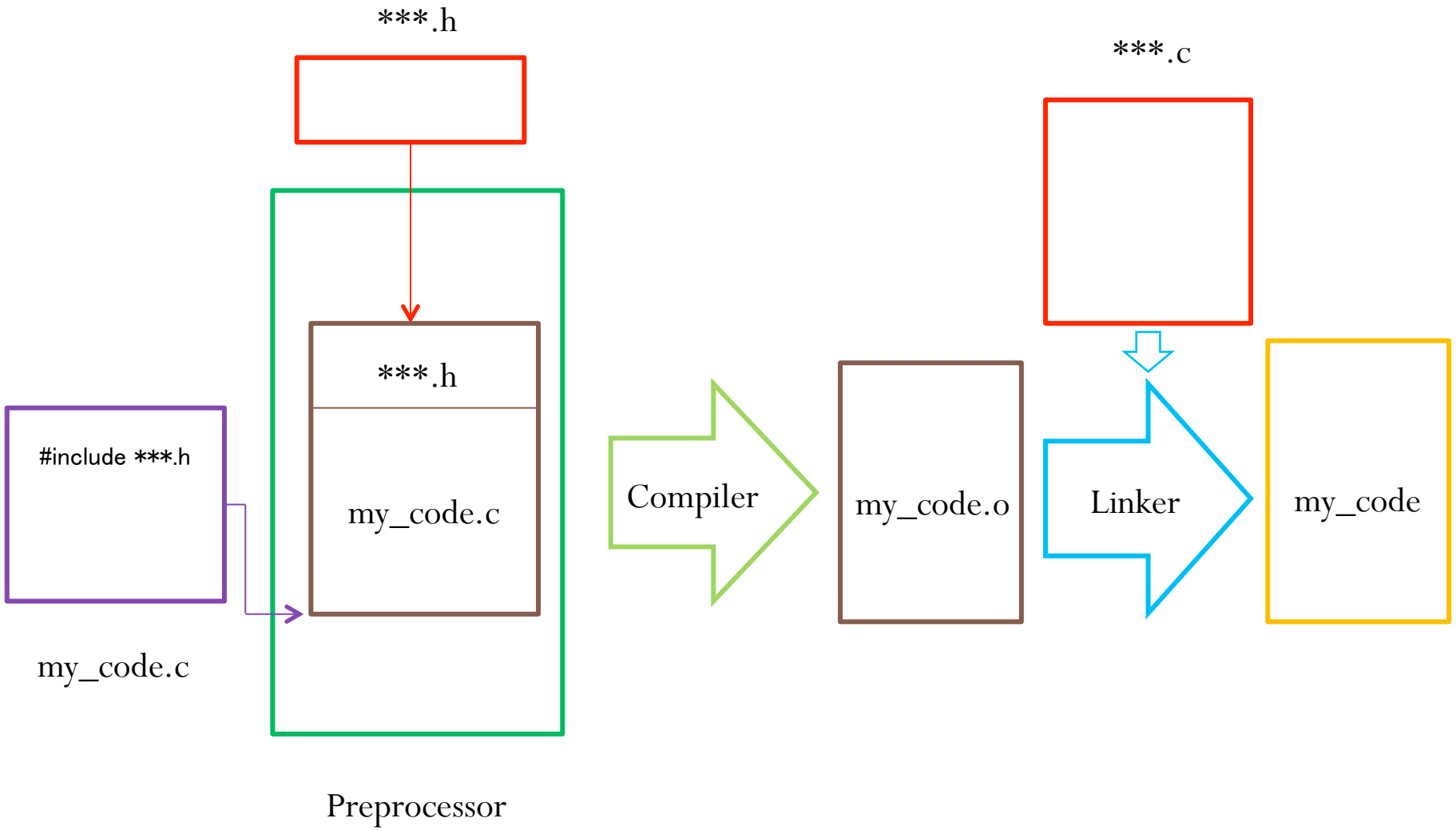
```
#include ″one.h″
#include ″two.h″
int z;
static int w;
void c (int d)
{   z= w − d; x= d;   }
```

Program partitioning
Header files

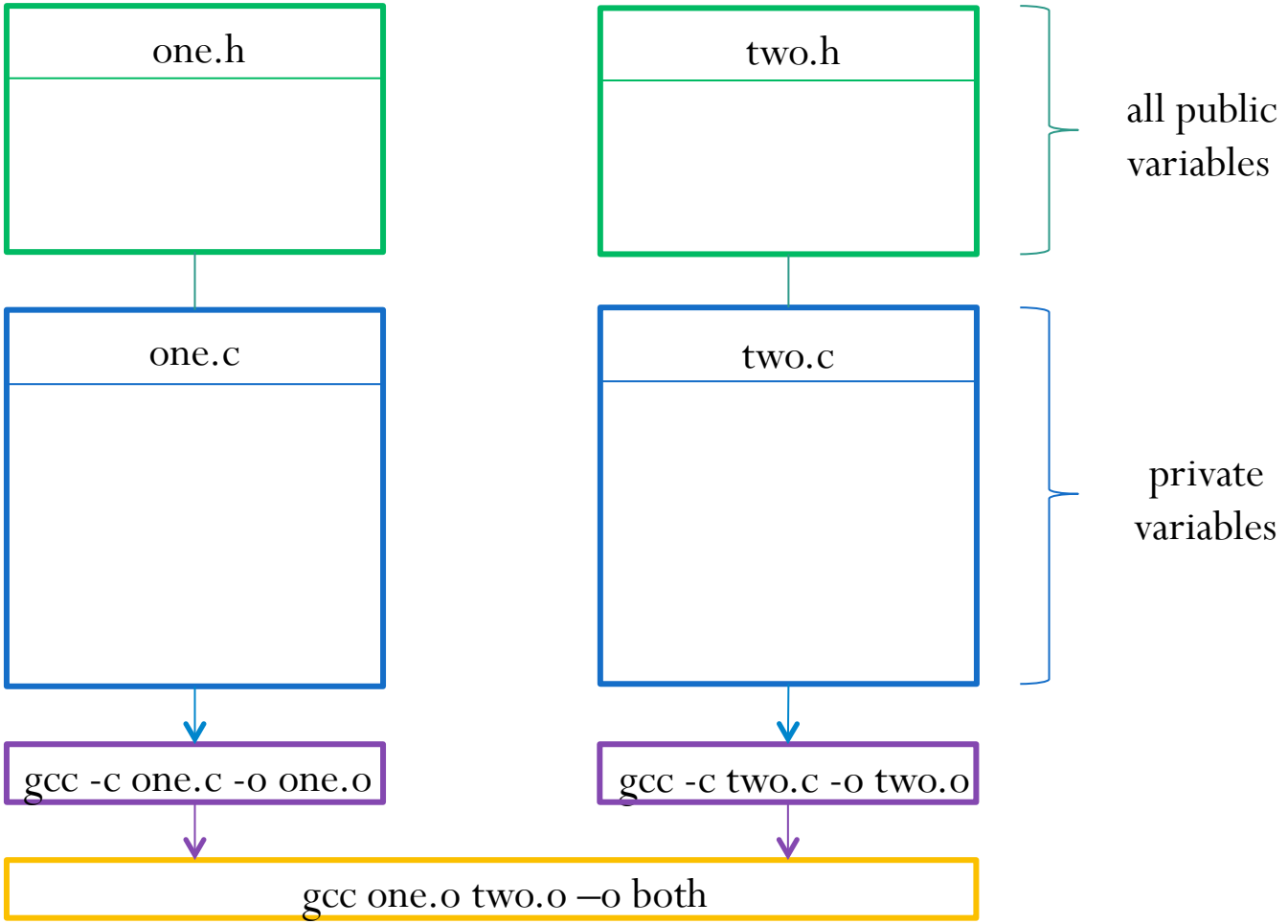
- Complex C programs usually consist of several files, which are compiled independently, and then linked to produce a single executable.
- Many useful functions (e.g. needed for input/output, mathematical calculations, etc.) are stored in the standard libraries, which is used by the pre-processor command #include<***.h>. What is the Java equivalent of this command? (import)
- This command indicates to pre-processor to include the file ***.h as a part of the source prior to compilation.
- The header files are not libraries of functions. Instead, they contain the function prototypes and global variable declarations for the code stored in the library, i.e. they serve as interfaces.
- What is the difference between the following two declarations?

#include <***.h> #include "***.h"

Program partitioning
Header files



Program partitioning
Independent compilation



Mistakes, mistakes...
Checking the execution of library functions

- If you call a library function (for example **malloc**, **fopen**, **fclose**, **sscanf**), you should always check whether it was successfully executed and trap any errors. Also, check whether the input value is correct.
- Example:

```
FILE *fPtr;
fPtr = fopen("file.c", "r");
if(!fPtr) { fprintf(stderr,' File not opened' ); exit(-1) }
```

- Example:

```
char *new_memory;
int memory_size = 20 * sizeof(char);
new_memory = (char *)malloc(memory_size);
if(new_memory == NULL) fprintf(stderr,"No memory could be allocated!
\n");
```

Mistakes, mistakes...
Checking the execution of library functions

- Always write error messages using stderr stream, rather than stdout (the former is not buffered, and the message appears on the screen immediately.
- The main program is declared as int for a reason – you should match any errors in the execution by returning a non-zero integer. However, you should have different integers for different sorts of errors (for example -1 for a file opening problem and -2 for memory allocation problem).
- Use exit command to terminate the program if an error is detected, rather than a long if–then–else construction.

```
if(error_condition)
   {error_message}
else
   {rest of the code}
```

```
if(error_condition)
   {error_message; exit}
```

Mistakes, mistakes...
Memory allocation

- You must allocate memory for an item before trying to access it.
- Every use of **malloc** should be paired with **free** (remember C does not have automatic garbage collection).
- You always need to cast a pointer returned from **malloc** before using it.
- After allocating memory, you are returned a pointer to it. Do not change it, otherwise, you will lose the allocated space, and/or the information it holds.

```
int* i=(int*)malloc(sizeof(int)); i=i1;
```

Mistakes, mistakes...
Pointers

- There is a general confusion about pointers.
  1. The use of unnecessary layers of pointers (for example, use a pointer to pointer, only to dereference it – so this is correct, but too complex and inefficient).
  2. The use of * operator for both declaring a pointer variable and dereferencing the pointer is a source of problems.
  3. Confusing the pointer value and the value of the location it points to (try to draw a diagram of what you want to do on a piece of paper before writing the code). Make sure that you understand the difference between, for example, `*p` and `&p`, or `(*p).name` and `p->name`.

Mistakes, mistakes...

Miscellaneous

- Make sure that you end your `typedef` declaration by a semi-column. If not, this can introduce errors further down the code, and it is not obvious what is the cause.
- If using math functions, you need the flag `–lm` when linking the code (always check your makefile).
- There is a difference between an array and a linked list. You cannot access immediately any element of the list, instead you need to iterate through it.
- When copying strings, it is good practice to use the library functions `strcpy` or `strdup`, rather than an assignment.