The C programming Language

Cosmin Bonchis
cosmin.bonchis@e-uvt.ro

Organizational

- Course objectives: present the C programming language, and programming concepts
- Grading:
  - 50% laboratory: (! avg 5 mandatory for exam acceptance)
    - Activity on class,
    - Homeworks,
    - Theoretical quizzes.
    - Laboratory exam
  - 50% exam:
    - exam quiz (mandatory 5! for next step),
    - exam programming oral assignments

Important

- **Lecture attendance**: required.
- **Expect you**: To be up to date with class material. To hand out programming assignments by the stated deadlines.
- **Expect you**: Work hard.
- **Academic honesty**: cheating leads to failing class and reporting.OK/encouraged: speak up in class. Two-way, rather than one-way communication. Request: be concise, to the point.
- **Disclaimer**: I can make mistakes/be wrong. Let me know (in person, email) how I can improve things.

Resources:

Literature:

Books:

B. Kernighan, D. Ritchie - *The C Programming Language*, 2nd ed., Prentice-Hall,1988
Ivor Horton – Beginning C: From Novice to Professional
Steve Oualline - Practical C Programming, Third Edition

Online lectures:

C Programming. Brian Brown, Central Institute of Technology, NZ. Constantin quizzes
C Programming Steven Summit, Experimental College, University of Washington, USA.
Introduction to C Programming, University of Leicester, UK.
C Programming. Steve Holmes, University of Strathclyde, UK.
C Language Tutorial. Drexel University, USA. A short introduction

official documents:

ISO/IEC 9899:1990 *(the C90 standard)*
ISO/IEC 9899:1999 *(the C99 standard)*

on the web:

C-FAQ - http://www.eskimo.com/~scs/C-faq/top.html

Software:

Whatever ***ANSI/ISO standard-complying*** compiler (and library), standalone or IDE

E.g.*:*

- free:

gcc *(Linux)*+**Code::blocks** *as an IDE,* **MinGW GCC** *(Win32)* + **Code::blocks** *as an IDE,* **djgpp** *(DOS)* + **rhide** *as an IDE*

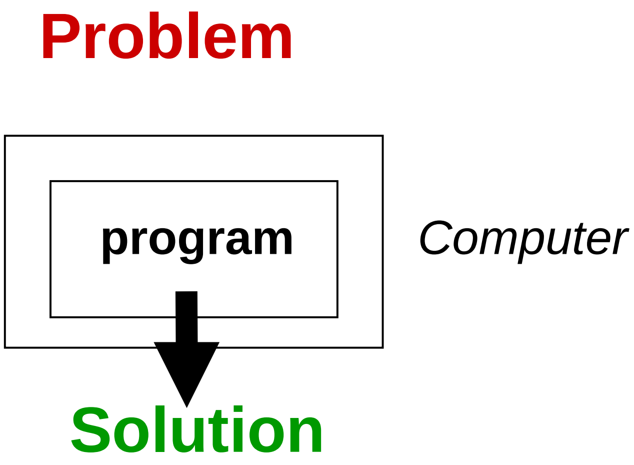
- commercial:

...

Communicating with computers is not easy !

It would be nice if we could write programs in English

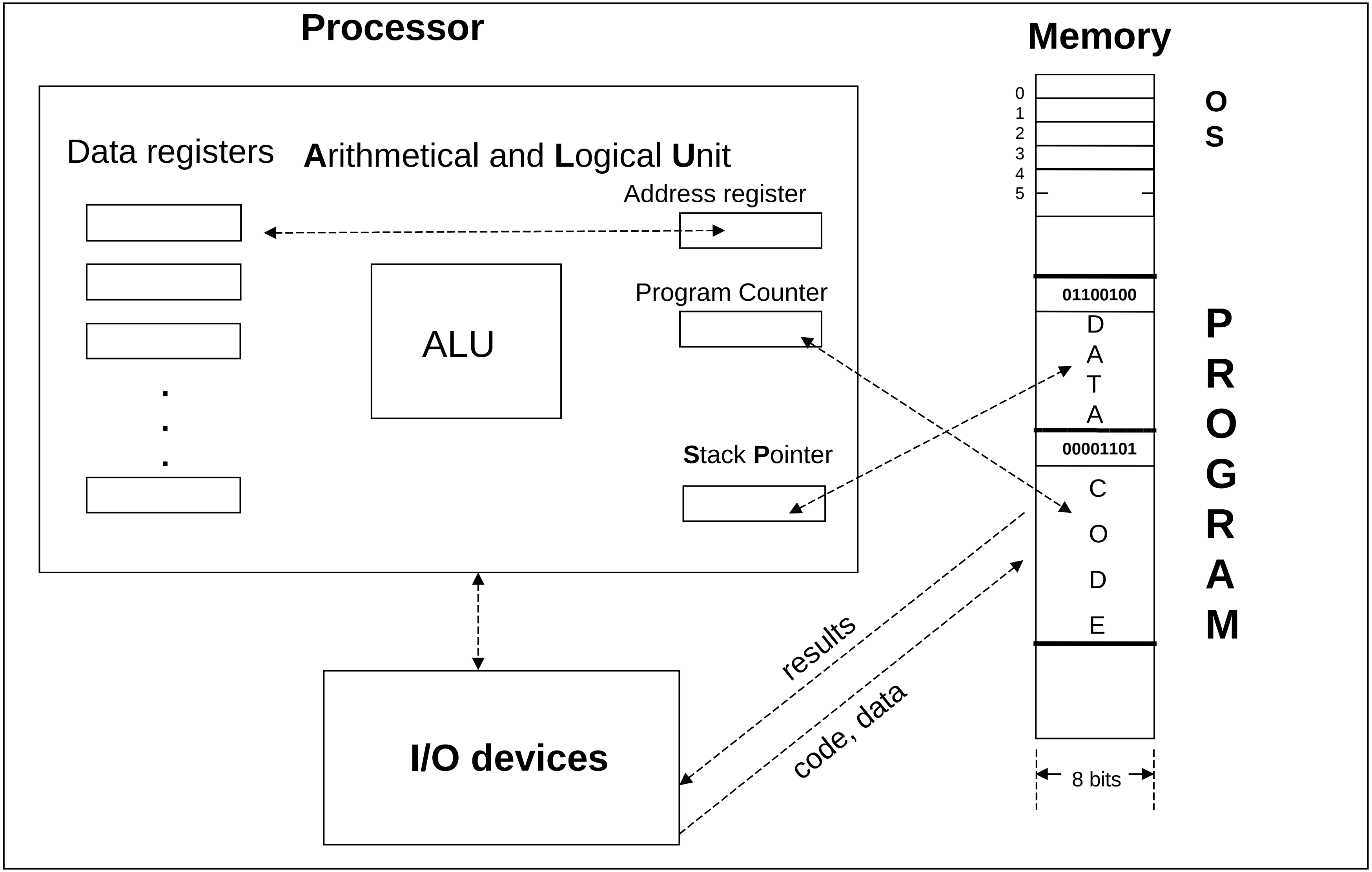
PROGRAM: *a concise definition*



Niklaus Wirth *(author of Pascal)*:

Program = Data + Algorithm

PROGRAM: *a concise definition - continued*



based on Lucian Cucu's lecture - The C

Programming Languages

Low level:

machine code language

assembly language (assembler)

C (early '70) by **D. Ritchie (Bell Labs)**

High level:

FORTRAN (early '50)
COBOL ('50)
LISP (late '50)
ALGOL (58, 60, 68)
PASCAL
Prolog (logical)
Smalltalk, C++, JAVA (OOP)
Haskell, Scheme (functional)
…

The C Programming Language

Developed for ***system programming*** (rewriting of UNIX OS for PDP-7 and PDP-11)

Later used also for general programming

First programming language implemented on almost all operating systems

First standardized programming language (ANSI C – 1989)

Characteristics:

- highly portable

- fast and compact code

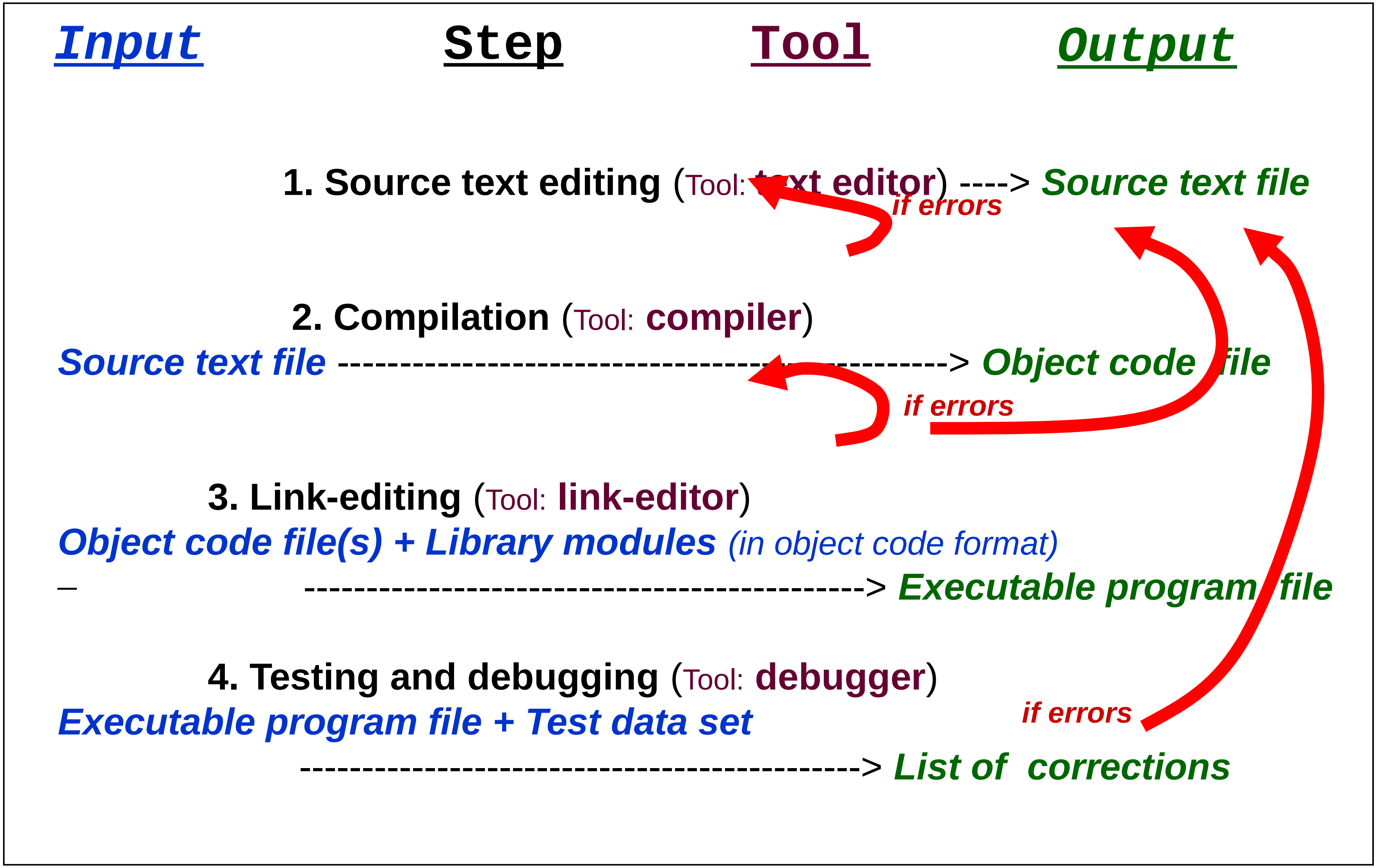
- small and flexible

- …

Best suited for

- system programming
- embedded programming

Basic steps in developing an application



Input **Step** **Tool** ***Output***

1. Source text editing (Tool: **text editor**) ----> ***Source text file***

if errors

2. Compilation (Tool: **compiler**)

Source text file --> ***Object code file***

if errors

3. Link-editing (Tool: **link-editor**)

Object code file(s) + Library modules *(in object code format)*

– --> ***Executable program file***

4. Testing and debugging (Tool: **debugger**)

Executable program file + Test data set

if errors

--> ***List of corrections***

Program structure: functions

C program = set of function definitions

+ declarations of functions and global variables

+ preprocessor directives

Why?

Real programs are very large and complex!

Developing them as a single functional unit is

- not practical
- not at all easy to maintain
- hard to reuse

Therefore, they are broken up in several (not seldom, hundreds or thousands of) smaller **functional units**, usually grouped, according to their functionality in separate *translation units* (**source files**).

Functional units:
- functions (in C programms: functions returning some value)
- procedures (in C programms: void functions)

Program structure: breaking down into *functions*

```
#include <stdlib.h>
#define N 1000
enum boolean { FALSE, TRUE};
int main()
{
int a[N], b[2*N], i, sorted=FALSE;
for(i=0;i<N; i++)      /* init a */
     a[i]=rand();
for(i=0;i<2*N; i++)           /* init b */
     b[i]=rand();
while(!sorted)         /*sort a */
     {
     sorted=TRUE;
     for(i=0; i<N-1;i++)
            if(a[i]>a[i+1])
                  {
                  int aux;
                  aux=a[i];
                  a[i]=a[i+1];
                  a[i+1]=aux;
                  sorted=FALSE;
                  }
     }
while(!sorted)        /*sort b */
     {
     sorted=TRUE;
     for(i=0; i<N-1;i++)
            if(b[i]>b[i+1])
                  {
                  int aux;
                  aux=b[i];
                  b[i]=b[i+1];
                  b[i+1]=aux;
                  sorted=FALSE;
                  }
     }
...
}
```

```
#include <stdlib.h>
#define N 1000
enum boolean { FALSE, TRUE};
int a[N], b[2*N];
int main()
{
 init_a();
 init_b();
 sort_a();
 sort_b();

}

void  init_a()
{
...
}
void init_b()
{
...
}
void sort_a()
{
...
}
void sort_b()
{
...
}
```

Program structure: parameterizing *functions*

Program.c

```
#include <stdlib.h>
#define N 1000
enum boolean { FALSE, TRUE};
int main()
{
int a[N], b[2*N], i, sorted=FALSE;
for(i=0;i<N; i++)         /* init a */
      a[i]=rand();
for(i=0;i<2*N; i++)       /* init b */
      b[i]=rand();
while(!sorted)            /*sort a */
      {
      sorted=TRUE;
      for(i=0; i<N-1;i++)
              if(a[i]>a[i+1])
                      {
                      int aux;
                      aux=a[i];
                      a[i]=a[i+1];
                      a[i+1]=aux;
                      sorted=FALSE;
                      }
      }
while(!sorted)           /*sort b */
      {
      sorted=TRUE;
      for(i=0; i<N-1;i++)
              if(b[i]>b[i+1])
                      {
                      int aux;
                      aux=b[i];
                      b[i]=b[i+1];
                      b[i+1]=aux;
                      sorted=FALSE;
                      }
      }
}
```

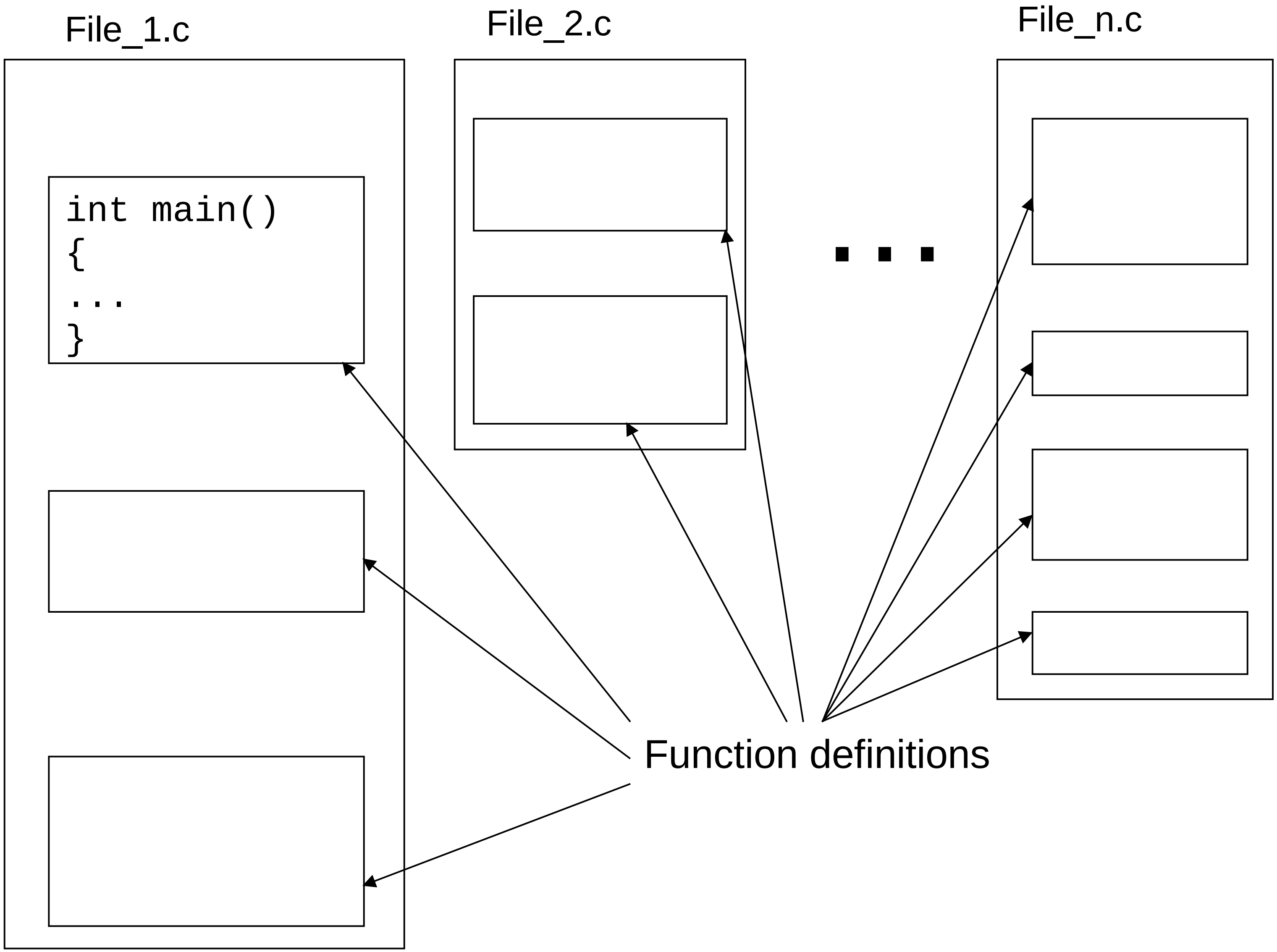
Main.c

```
#include <stdlib.h>
#define N 1000
enum boolean { FALSE, TRUE};

void init(int [], int);
void sort(int [], int);

int main()
{
int a[N], b[2*N];
init(a, N);
init(b, 2*N);
sort(a,N);
sort(b, 2*N);
}
```

Array_f.c

```
void init(int t[], int n)
{
int i;
for(i=0;i<n; i++)
        t[i]=rand();
}

void sort(int t[], int n)
{
int i, sorted=FALSE;
while(!sorted)      /*sort a */
        {
        sorted=TRUE;
        for(i=0; i<n-1;i++)
                if(t[i]>t[i+1])
                        {
                        int aux;
                        aux=t[i];
                        t[i]=t[i+1];
                        t[i+1]=aux;
                        sorted=FALSE;
                        }
        }
}
```

Program structure: translation units (source files)

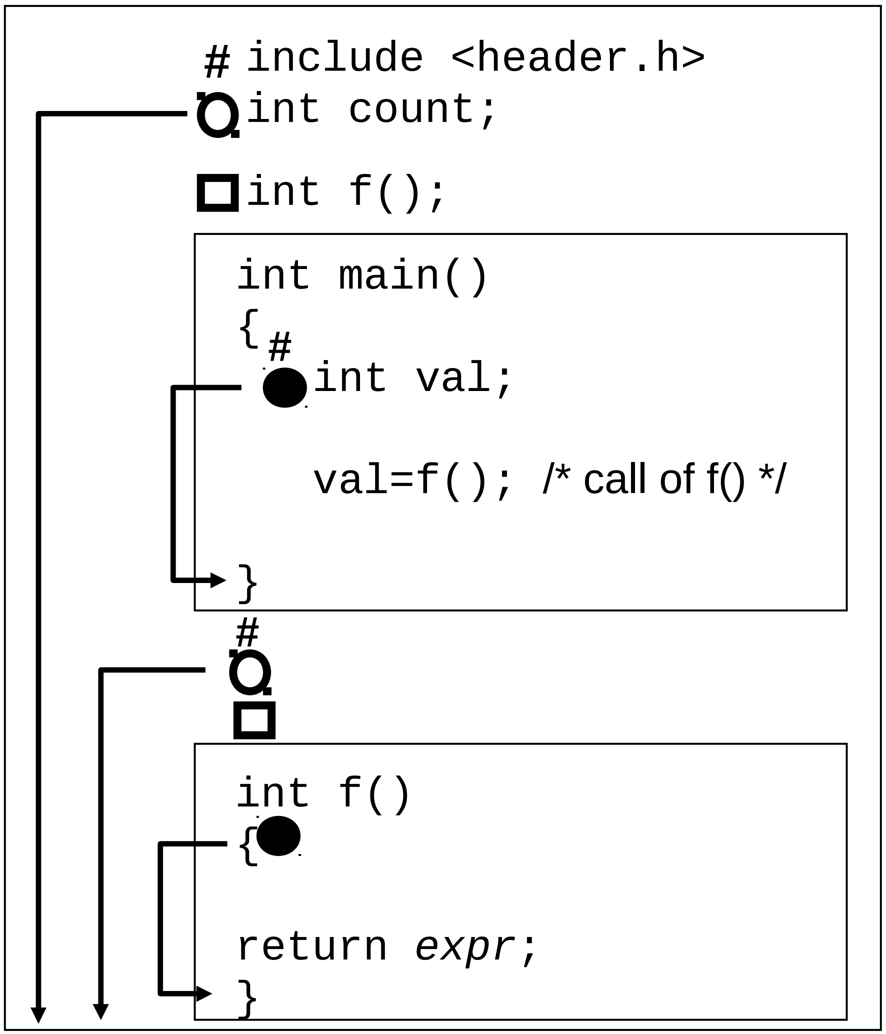


```
int main()
{
...
}
```

based on Lucian Cucu's lecture - The C

Program structure (refined)

file.c



```
# include <header.h>
int count;

int f();

int main()
{ #
    int val;

    val=f(); /* call of f() */

}
#


int f()
{

return expr;
}
```

Legend:

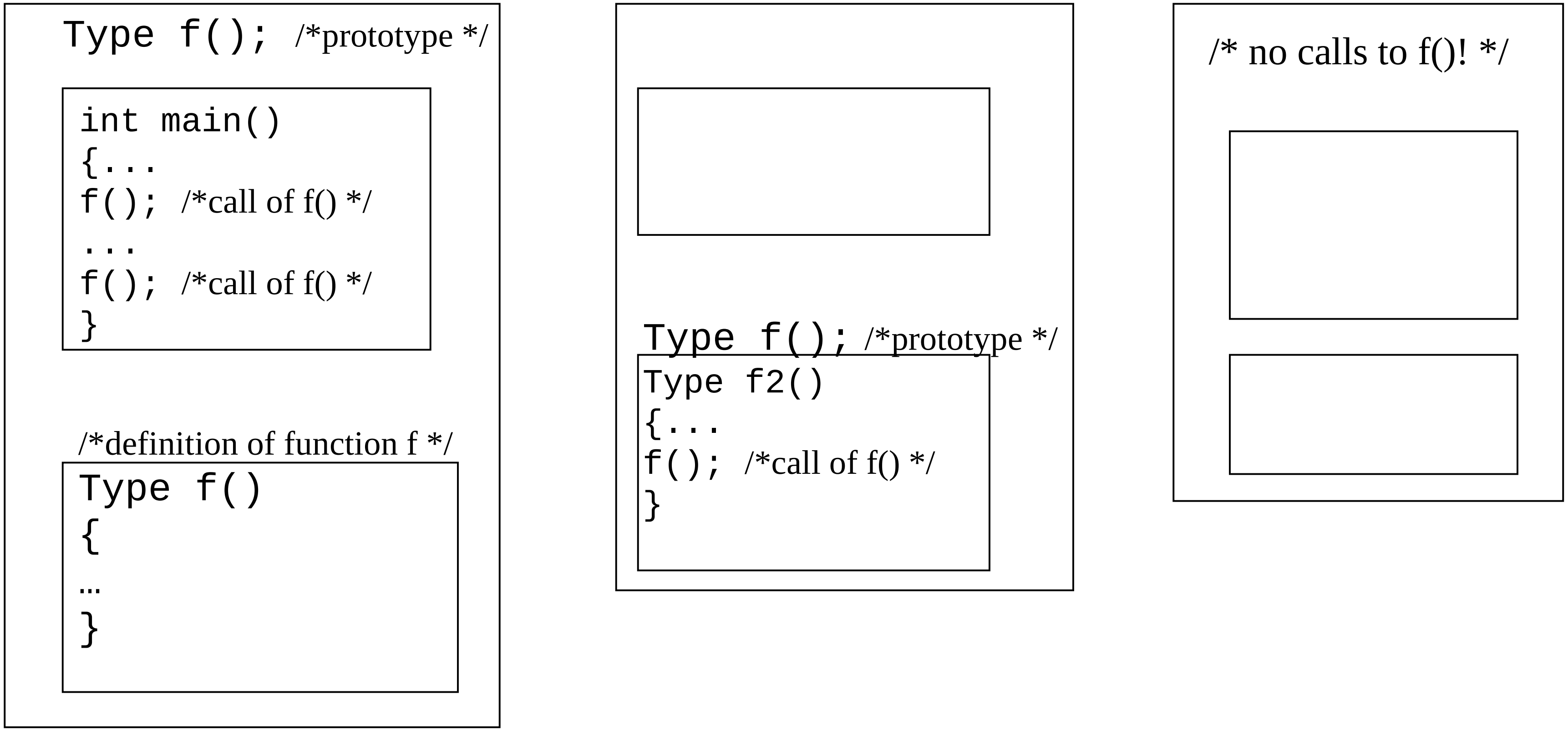
- # preprocessor directive
- ▯ declaration of global variable
- □ function declaration (prototype)
- ● declaration of local variable
- ▭ function definition
- ↳ scope of identifiers

A function is executed only if it is called!

Program structure: functions

Function:

- declaration (*prototype*) – in each translation unit where a call exists
- calls – several, even in the same translation unit
- definition – unique throughout all translation units



```
Type f(); /*prototype */

int main()
{...
f(); /*call of f() */
...
f(); /*call of f() */
}

/*definition of function f */
Type f()
{
…
}
```

```
Type f(); /*prototype */
Type f2()
{...
f(); /*call of f() */
}
```

```
/* no calls to f()! */
```

Program structure: functions

Function declaration:

Type *function_name* (**<parameter type_declaration_list>** *);*

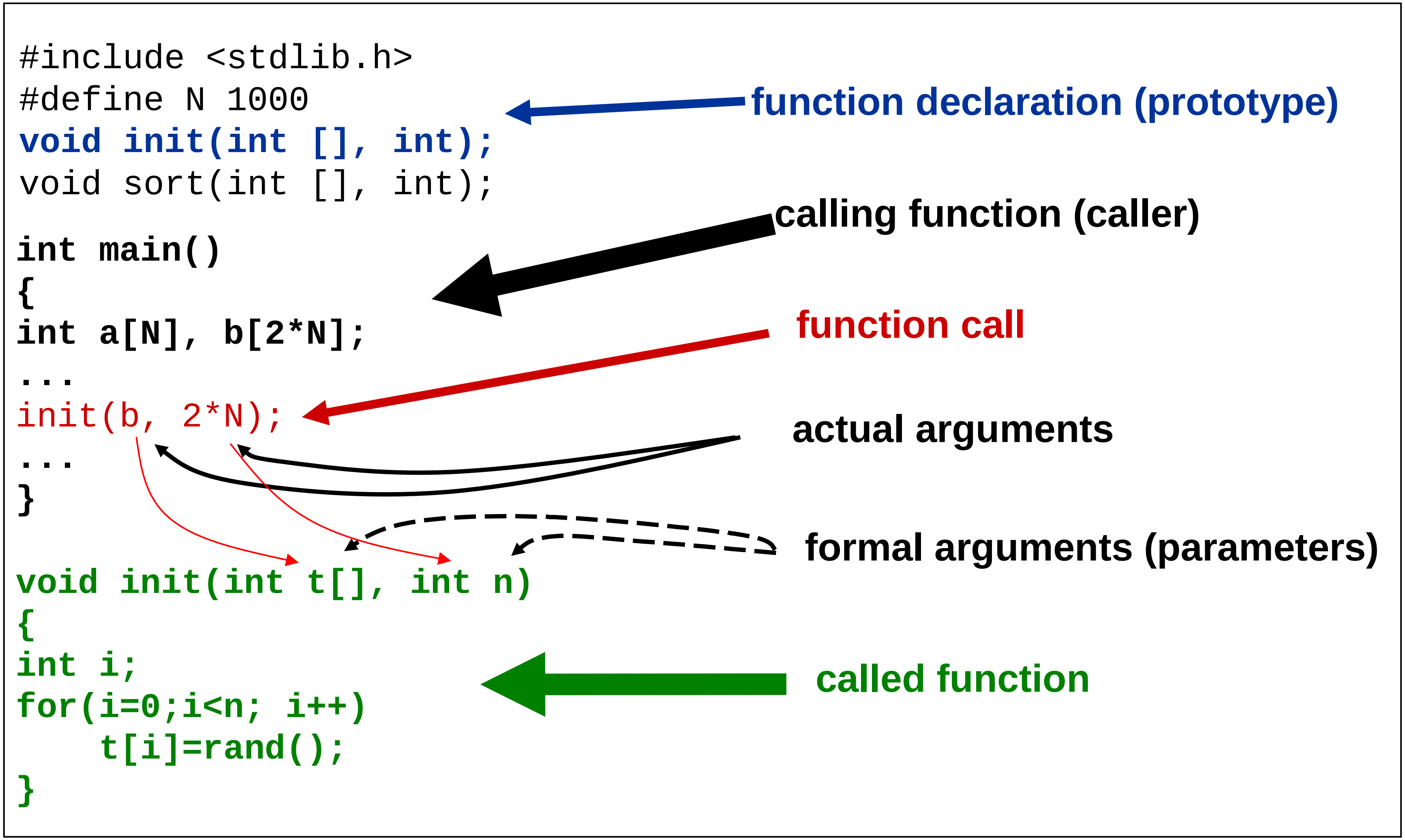
Function definition:

Type *function_name* (**<parameter declaration_list>** *)*
{
*/*declarations of local variables and functions*/*
/ statements*/*
…
return *expression;* */* expression* of type **Type** **/*
}

Function call:

function_name (**<actual_argument_list>** *);*
var= function_name (**<actual_argument_list>** *);*

Functions: taxonomy



```
#include <stdlib.h>
#define N 1000
void init(int [], int);
void sort(int [], int);

int main()
{
int a[N], b[2*N];
...
init(b, 2*N);
...
}

void init(int t[], int n)
{
int i;
for(i=0;i<n; i++)
    t[i]=rand();
}
```

function declaration (prototype)

calling function (caller)

function call

actual arguments

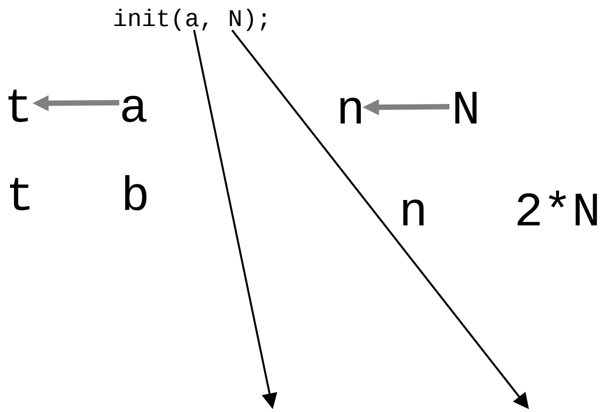
formal arguments (parameters)

called function

Program structure: means of *communication between functions*

Communication: sharing data

- through the actual arguments,
- through the returned value
- through global variables



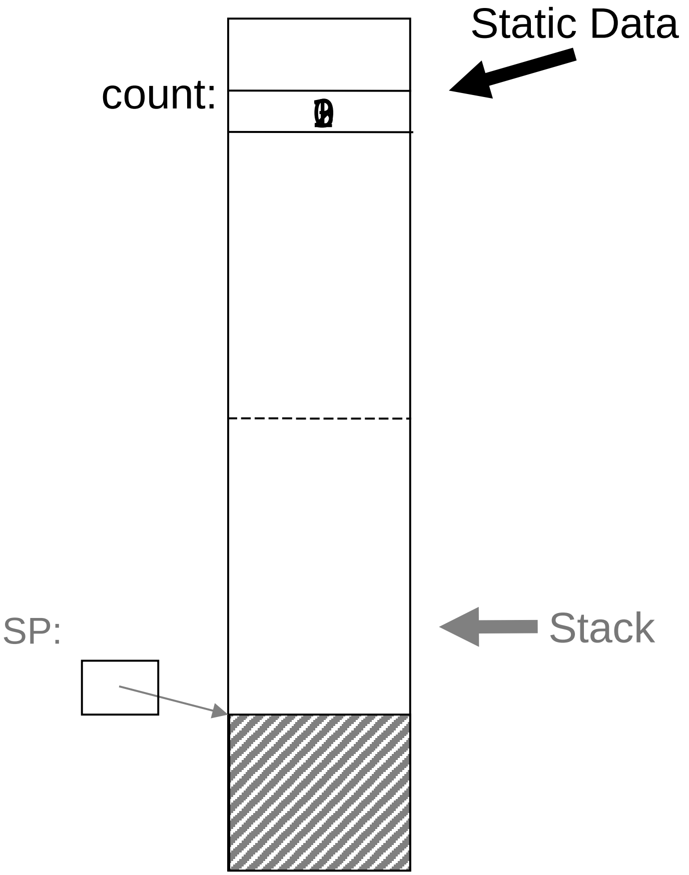
```
void init(int t[], int n)
{
int i;
for(i=0;i<n; i++)
    t[i]=rand();
}
```

```
#include <stdlib.h>
#define N 1000
enum boolean { FALSE, TRUE};

void init(int [], int);
void sort(int [], int);

int main()
{
int a[N], b[2*N];
init(a, N);
init(b, 2*N);
sort(a,N);
sort(b, 2*N);
}

void init(int t[], int n)
{
int i;
for(i=0;i<2*n; i++)
       t[i]=rand();
}

void sort(int t[], int n)
{
int i, sorted=FALSE;
while(!sorted)           /*sort a */
       {
       sorted=TRUE;
       for(i=0; i<n-1;i++)
              if(t[i]>t[i+1])
                     {
                     int aux;
                     aux=t[i];
                     t[i]=t[i+1];
                     t[i+1]=aux;
                     sorted=FALSE;
                     }
       }
}
```

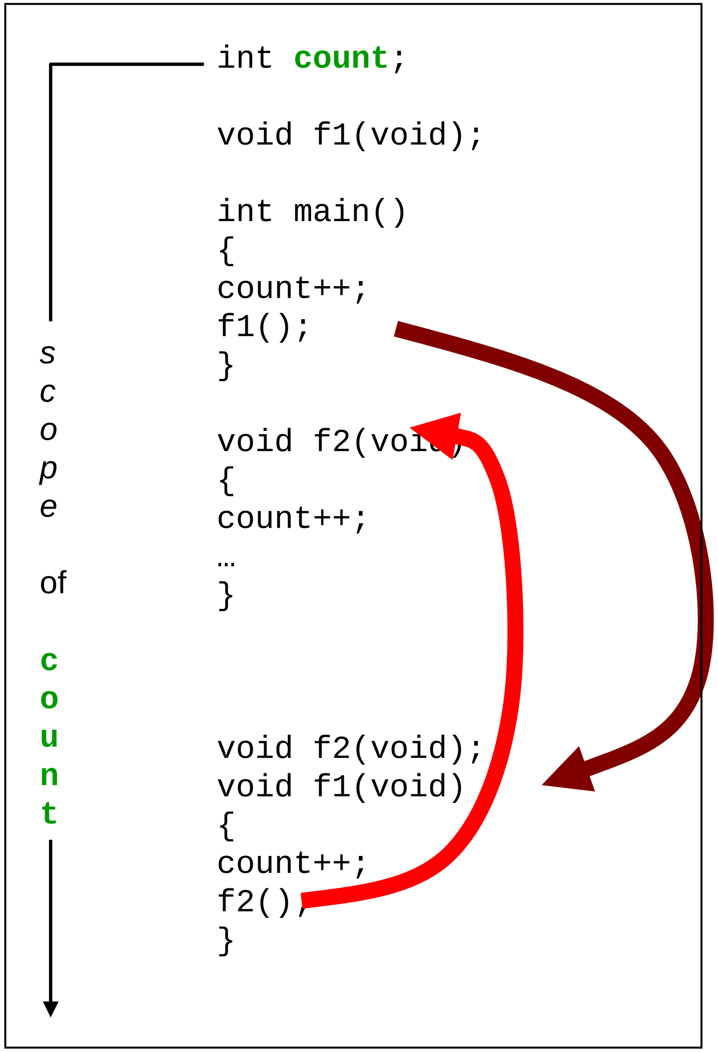
Program structure: means of *communication between functions*

Communication: sharing data

- through global variables



file.c



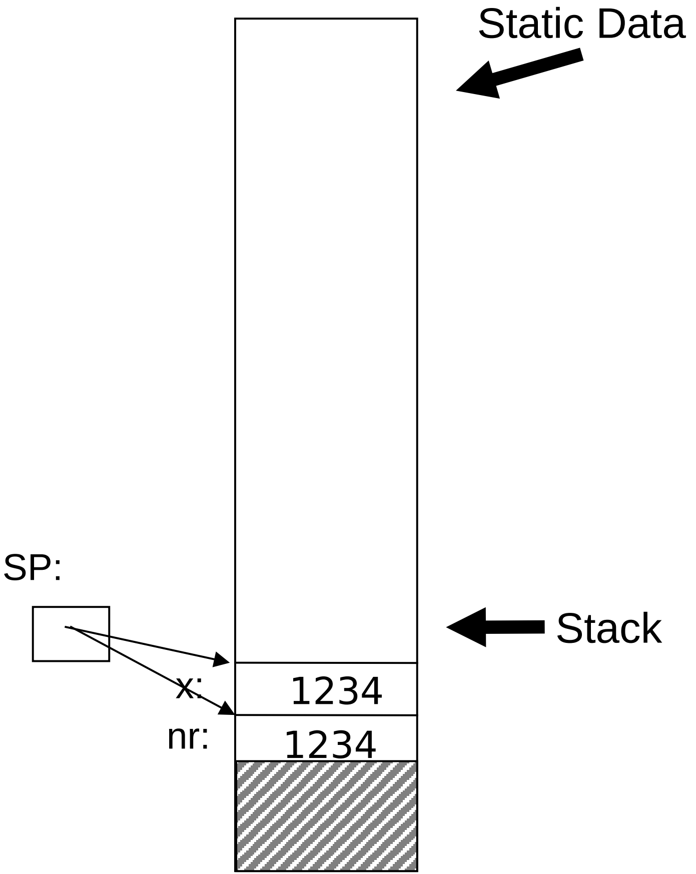
```
int count;

void f1(void);

int main()
{
count++;
f1();
}

void f2(void)
{
count++;
…
}



void f2(void);
void f1(void)
{
count++;
f2();
}
```

Program structure: means of *communication between functions*

Communication: sharing data

- through the returned value

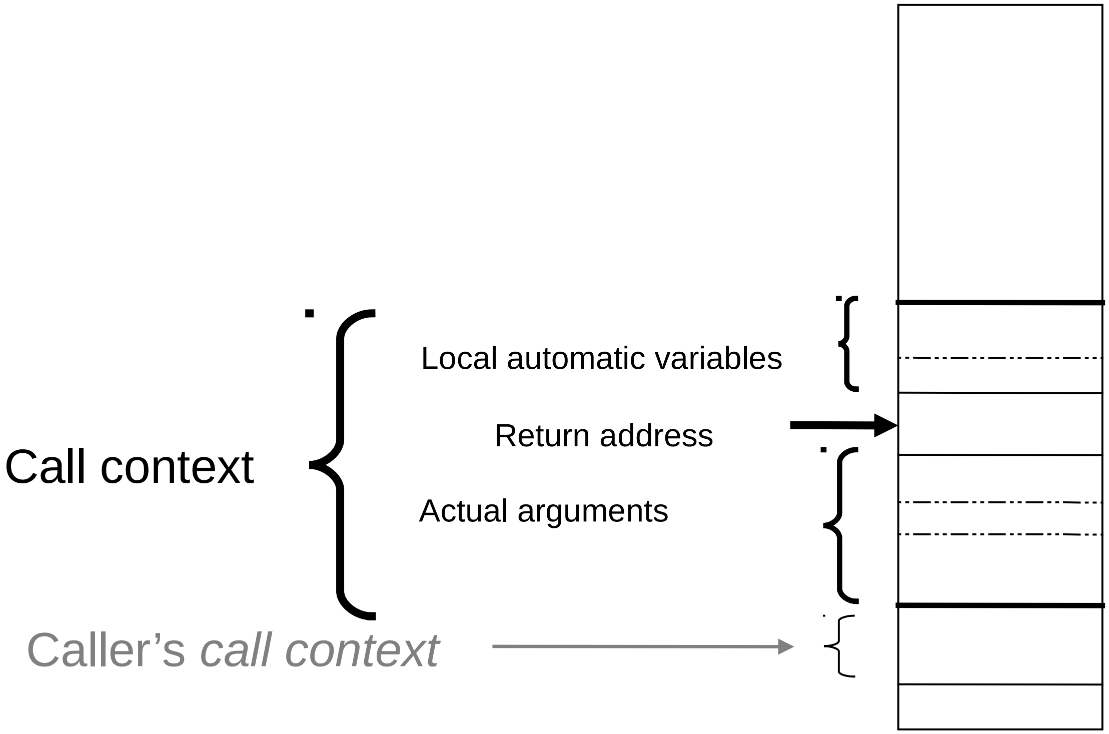
```
#include <stdio.h>
int getint();

int main() {
int nr;
 nr =  getint();
}

int getint()
{
int x;
scanf(“%d”, &x);
return x;
}
```



Functions: call context

The **call context** is a storage area on the stack on which:

- the values of actual arguments are copied (in reverse order!)
- the return address is saved
- the local automatic variables of the called function are created



Functions: The C calling convention

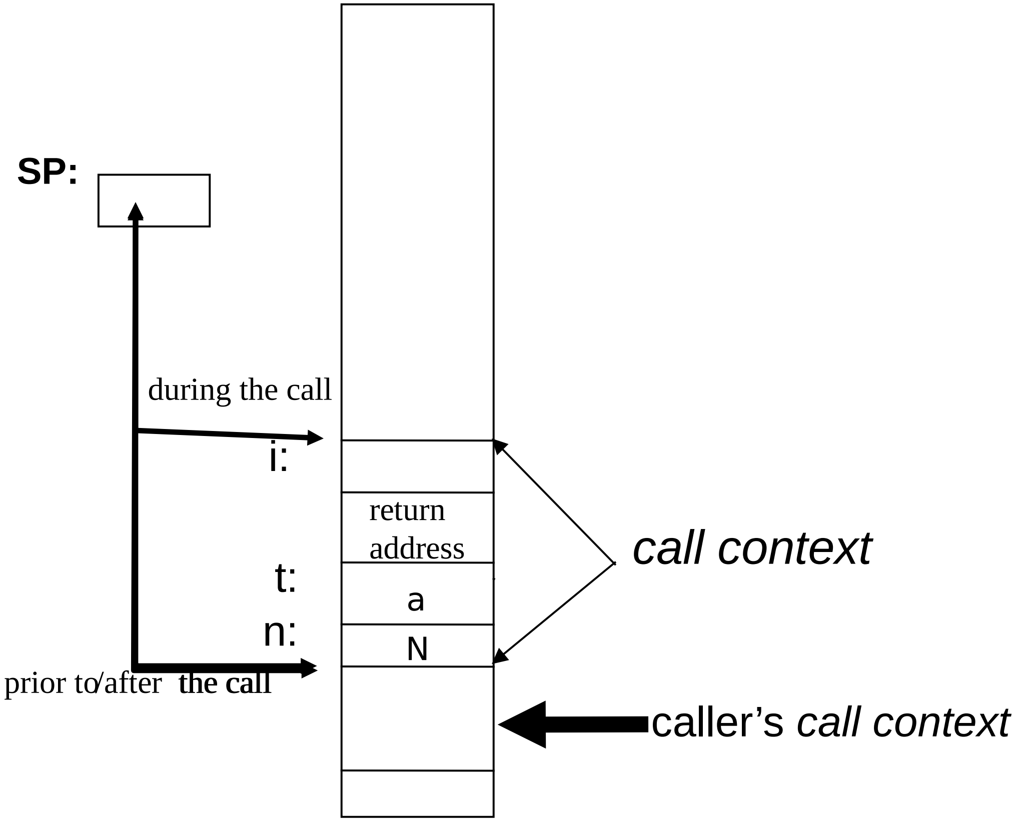
The C calling convention:

Arguments are passed (by the *calling function*) to the *called function*

- by value and
- in reverse order

```
...
init(a, N);
...
```

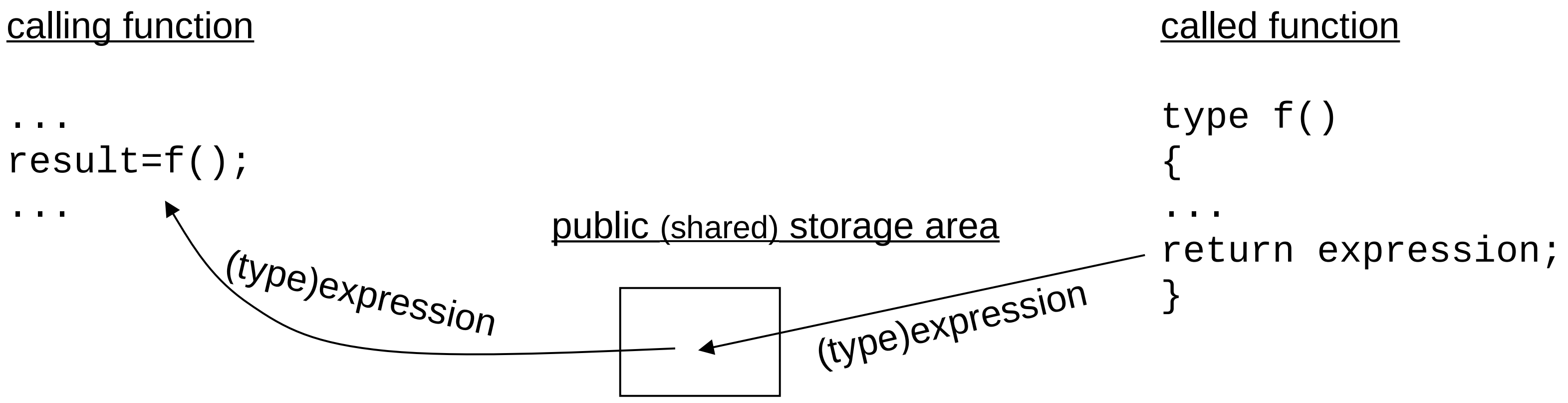
```
void init(int t[], int n)
{
int i;
for(i=0;i<n; i++)
    t[i]=rand();
}
```



Functions: the return mechanism

The return mechanism is implemented as a type-driven protocol, used by the compiler both when compiling:

- the *return expression* statement and
- the statement which takes over of the return value in the caller.



calling function

```
...
result=f();
...
```

called function

```
type f()
{
...
return expression;
}
```

Return mechanism: a possible protocol

The **return value**:

- is placed in a *register* (or several registers) by the called function
- is “taken” by the caller function from the *same register* (registers)

Exception:

if the return value is a structure, the address of a memory area where the actual structure is saved, is passed via registers to the caller function.

Return type	Public (shared) storage area
`char`	low byte of register R0
`short`	R0
`int (2 bytes)`	R0
`long`	R0, R1
`float`	R0, R1
`double`	R0, R1, R2, R3
`long double`	R0, R1, R2, R3, R4

Functions: recursive functions

Recursive function: a function that calls itself, directly or indirectly

Pro's:

"recursive code is **more compact**, and often much **easier to write and understand** than the non-recursive equivalent. Recursion is especially convenient for recursively defined data structures like trees." *(K&R, 4.10)*

Con's:

recursive functions need **more storage** area and take **more time to execute** than the non-recursive equivalent, because of the additional overhead incurred by the repeated function calls.

Caution:

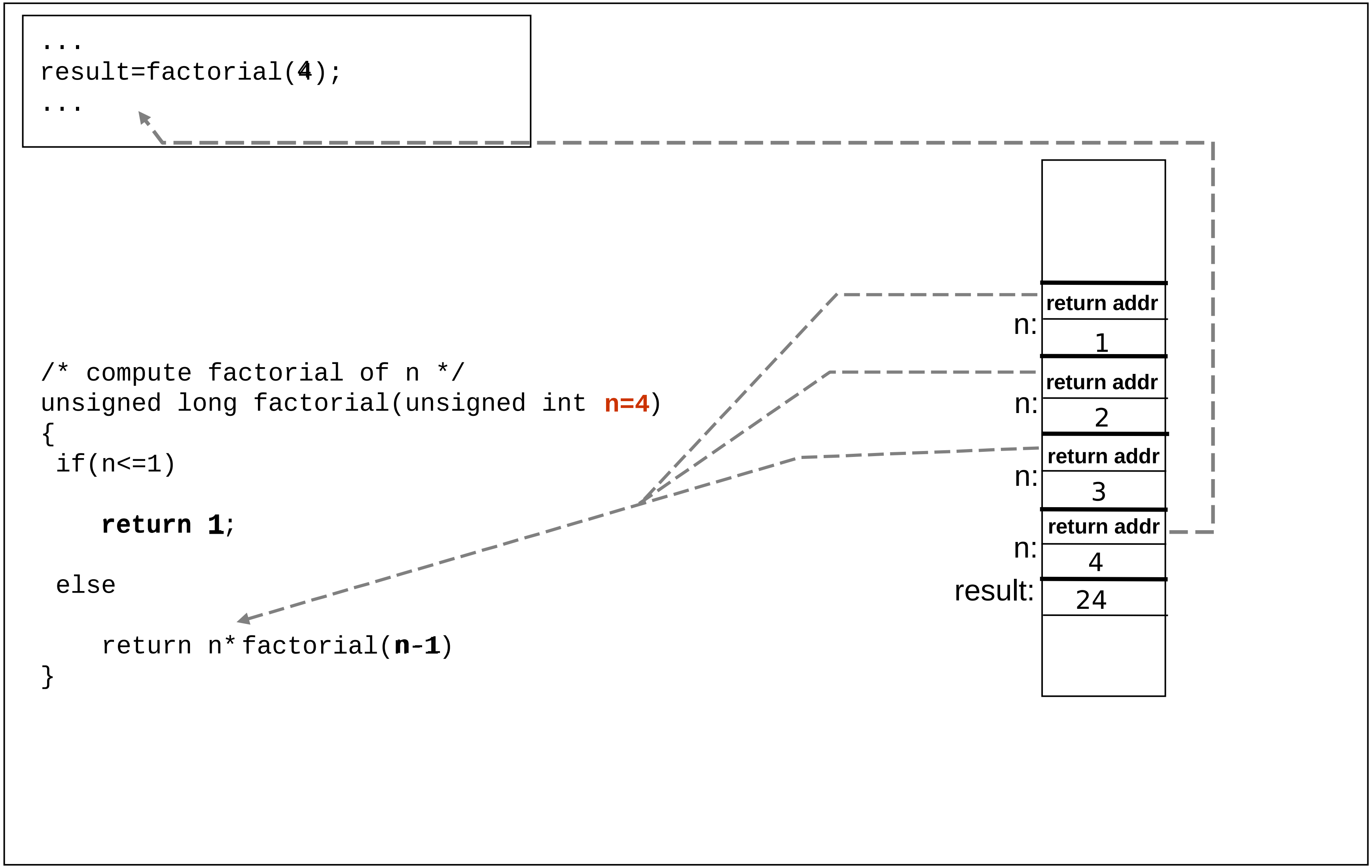
every recursive function has to test a condition to stop recursive calls to stand forever!

Recursive functions: some examples

```
 #include <stdio.h>
/* printd:  print n in decimal */
void printd(int n)
{
 if (n < 0)
    {
     putchar('-');
     n = -n;
    }
 if (n / 10)
    printd(n / 10);

/* if FALSE -> stop recursive calls! and print out a digit */
 putchar(n % 10 + '0');
}
```

```
/* compute factorial of n */
unsigned long factorial(unsigned int n)
{
 if(n<=1)       /* if FALSE stop recursive calls*/
    return 1;       /* and return to previous call */
 else
    return n*factorial(n-1);
}
```

Functions: recursive calls



```
...
result=factorial(4);
...
```

```
/* compute factorial of n */
unsigned long factorial(unsigned int n=4)
{
 if(n<=1)

    return 1;

 else

    return n*factorial(n-1)
}
```

Recursive functions: possible problems

If

- the recursive call never ends (missing a proper condition!)

or

- the recursive call is performed a large number of times

then, as a consequence, the stack may be exhausted!

In such cases, if

- the code was compiled with the compiler switch "check stack overflow"

  the program stops with the error message: "Stack overflow!"

- the code was compiled without the compiler switch "check stack overflow"

  the result is impressible!!!

Then, if recursion is less efficient and possibly dangerous,

why use recursion?

Because:

- it is easier to implement
- a certain type of recursive functions (tail-recursive functions) are automatically transformable to their iterative equivalent, which is more efficient!