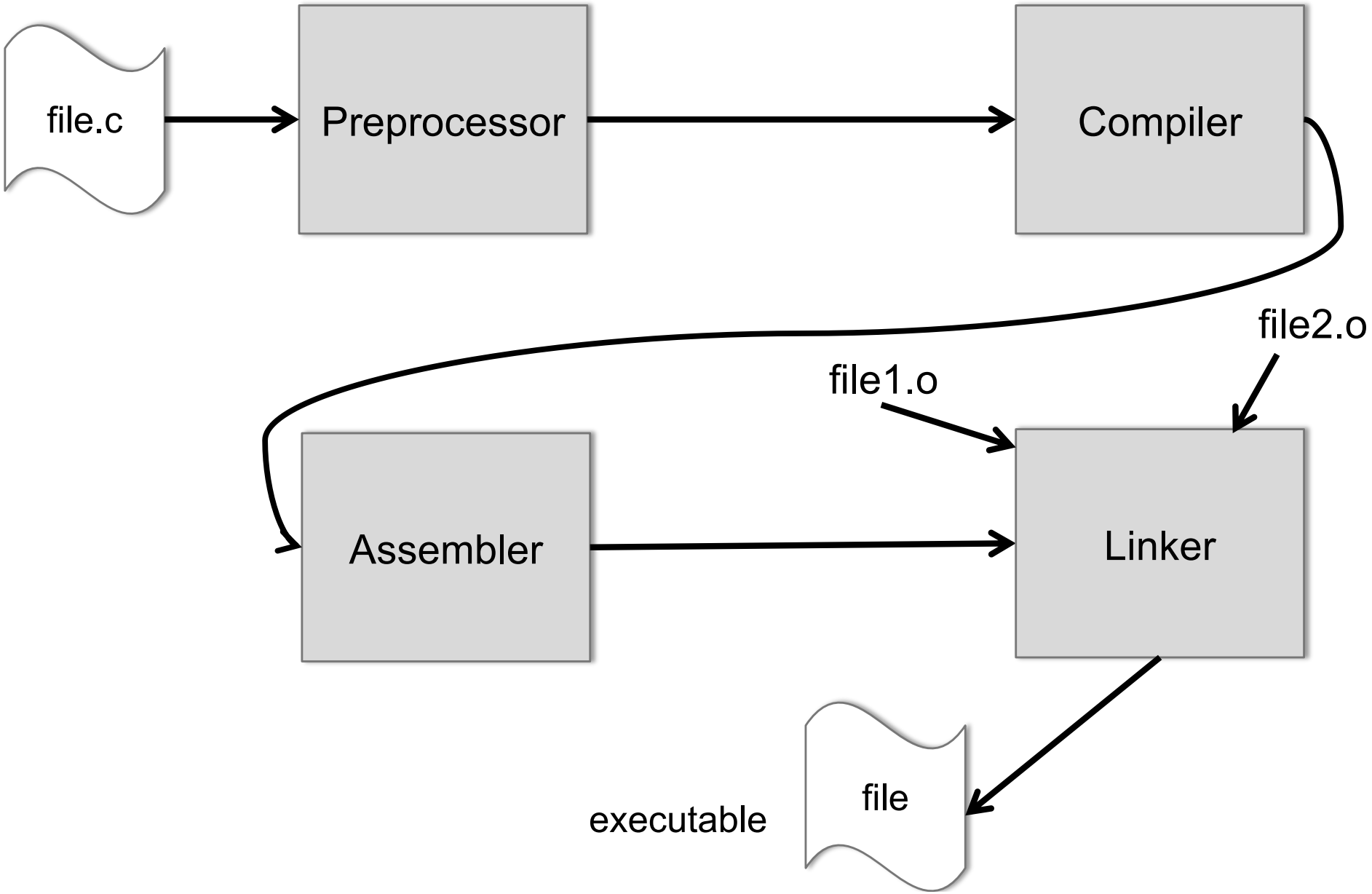
PROGRAMMAZIONE PROCEDURALE

A.A. 2023/2024

STEPS OF GCC

STEPS


file.c
Preprocessor
Compiler
file2.o
file1.o
Assembler
Linker
executable
file

PREPROCESSOR

PREPROCESSOR

- The C preprocessor is a *macro processor* that is used automatically by the C compiler to transform your program before actual compilation. It is called a macro processor because it allows you to define *macros*, which are brief abbreviations for longer constructs.
- The C preprocessor provides four separate facilities that you can use as you see fit:
- **Inclusion of header files.** These are files of declarations that can be substituted into your program.
- **Macro expansion.** You can define *macros*, which are abbreviations for arbitrary fragments of C code, and then the C preprocessor will replace the macros with their definitions throughout the program.
- **Conditional compilation.** Using special preprocessing directives, you can include or exclude parts of the program according to various conditions.

INCLUDE

- An #include directive instructs the preprocessor to insert the contents of a specified file in the place of the directive.
  - ✓#include <*filename*>
  - ✓ #include "*filename*"
- Use the first form, with angle brackets, when you include standard library header
  - ✓#include <math.h> // Prototypes of mathematical functions,
    // with related types and macros.
- Use the second form, with double quotation marks, to include source files specific to your programs.
  - ✓#include "myproject.h"

WHERE TO FIND HEADER FILES

- For files specified between angle brackets (<*filename*>), the preprocessor usually searches in certain system directories, such as */usr/local/include* and */usr/include* on Unix systems, for example.
- For files specified in quotation marks ("*filename*"), the preprocessor usually looks in the current directory first, which is typically the directory containing the program's other source files.
- If such a file is not found in the current directory, the preprocessor searches the system *include* directories as well.
- A *filename* may contain a directory path. If so, the preprocessor looks for the file only in the specified directory.

DEFINING AND USING MACROS

- You can define macros in C using the preprocessor **directive** #define.
- A common use of macros is to define a name for a numeric constant:

```
✓#define ARRAY_SIZE 100
 double data[ARRAY_SIZE];
```

ARRAY_SIZE is replaced with 100

- The preprocessor *expands* the macro; that is, it replaces the macro name with the text it has been defined to represent

USING MACROS WITH MACROS

- No macro can be expanded recursively

```
#define PI      3.141593
#define A       (A / 8)
```

PREPROCESSING

- Before submitting the source code to the actual compiler, the preprocessor remove comments, executes directives and expands macros in the source files.
- GCC ordinarily leaves no intermediate output file containing the results of this preprocessing stage.
- However, you can save the preprocessor output for diagnostic purposes by using the -E option, which directs GCC to stop after preprocessing.
- The preprocessor output is directed to the standard output stream, unless you indicate an output filename using the -o option:
  - ✓**gcc -E -o mylibrary.i mylibrary.c**

COMPILER

ASSEMBLY

- At the heart of the compiler's job is the translation of C programs into the machine's **assembly language**.
- Assembly language is a "human-readable" programming language that correlates closely to the actual machine code.
- Consequently, there is a different assembly language for each CPU architecture.
- Ordinarily GCC stores its assembly-language output in temporary files, and deletes them immediately after the assembler has run.

-S

- You can use the -S option to stop the compiling process after the assembly-language output has been generated.
- If you do not specify an output filename, GCC with the -S option creates an assembly-language file with a name ending in *.s* for each input file compiled.
- **gcc –S file.c**
- The compiler preprocesses *file.c* and translates it into assembly language, and saves the results in the file *file.s*.

EXAMPLE

```
int main()
{
   int a= 5;
   a++;
}
```

```
 .section        __TEXT,__text,regular,pure_instructions
        .macosx_version_min 10, 11
        .globl  _main
        .align  4, 0x90
_main:                                  ## @main
        .cfi_startproc
## BB#0:
        pushq   %rbp
Ltmp0:
        .cfi_def_cfa_offset 16
Ltmp1:
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
Ltmp2:
        .cfi_def_cfa_register %rbp
        xorl    %eax, %eax
        movl    $5, -4(%rbp)
        movl    -4(%rbp), %ecx
        addl    $1, %ecx
        movl    %ecx, -4(%rbp)
        popq    %rbp
        retq
        .cfi_endproc
.subsections_via_symbols
```

ASSEMBLER

ASSEMBLER

- Because each machine architecture has its own assembly language, GCC invokes an assembler on the host system to translate the assembly-language program into executable binary code.
- The result is an *object file*, which contains the machine code to perform the functions defined in the corresponding source file, and also contains **a *symbol table* describing all objects in the file that have external linkage**.

COMPILING AND LINKING SEPARAT.

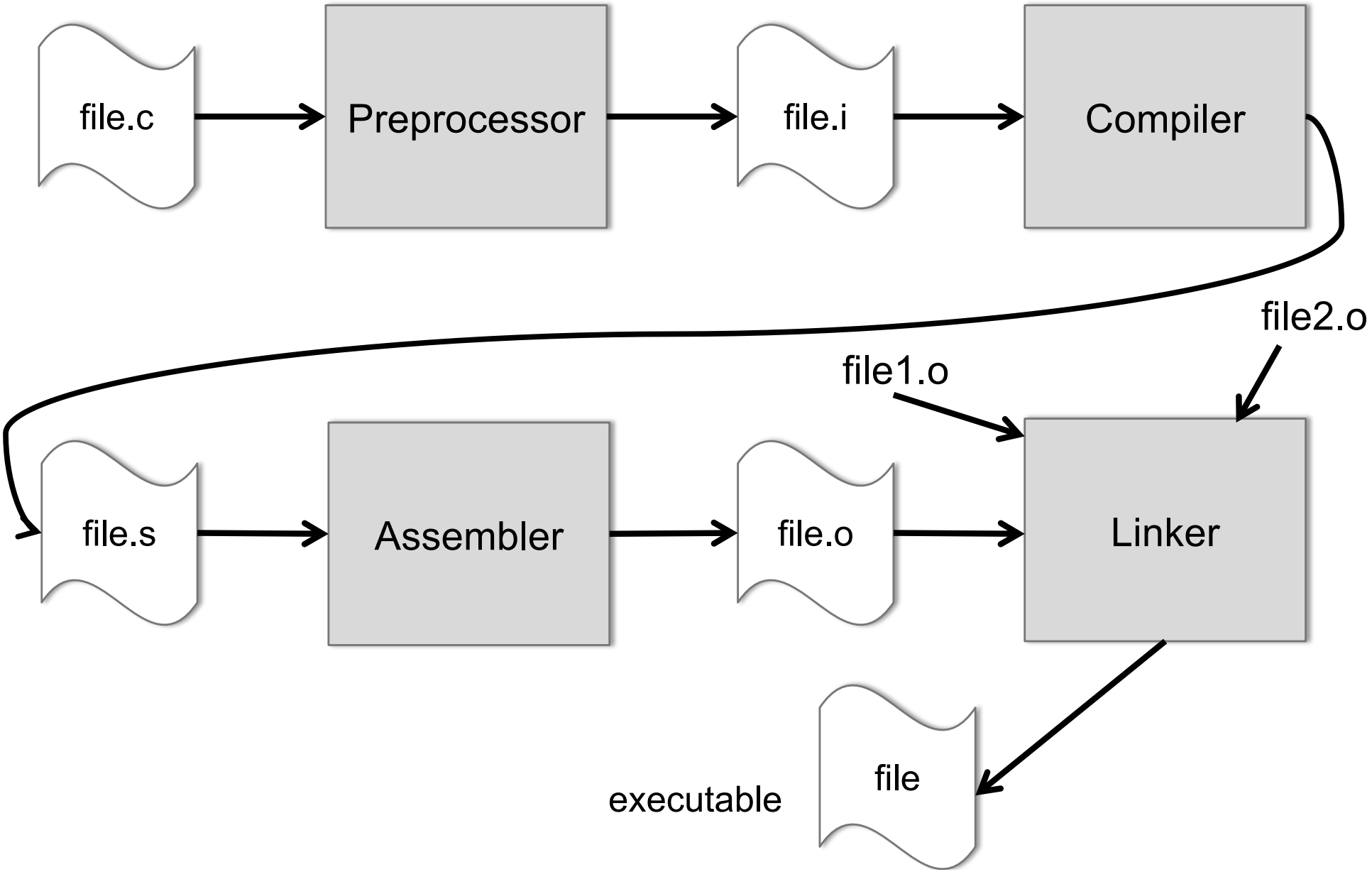
- If you invoke GCC to compile and link a program in one command, then its object files are only temporary, and are deleted after the linker has run.
- Most often, however, compiling and linking are done separately. The -c option instructs GCC not to link the program, but to produce an object file with the filename ending *.o* for each input file:
- $ **gcc –c mylibrary.c**

LINKING

LINKING

- The linker joins a number of binary object files into a single executable file.
- In the process, it has to complete the external references among your program's various modules by substituting the final locations of the objects for the symbolic references.
- The linker does this using the same information that the assembler provides in the symbol table.
- Furthermore, the linker must also add the code for any C standard library functions you have used in your program.

STEPS


file.c
Preprocessor
file.i
Compiler
file2.o
file1.o
file.s
Assembler
file.o
Linker
executable
file

GDB

GDB

- If the GNU C compiler, GCC, is available on your system, then GDB is probably already installed as well.
  - ✓gdb -version

```
GNU gdb (GDB) 7.10
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin15.0.0".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
```

SYMBOLS

- GDB is a symbolic command line debugger. “Symbolic” here means that you can refer to variables and functions in the running program by the names you have given them in your C source code.
- In order to display and interpret these names, the debugger requires information about the types of the variables and functions in the program, and about which instructions in the executable file correspond to which lines in the source files.
- a *symbol table*, which the compiler and linker include in the executable file when you run GCC with the -g option:
  - ✓gcc –g -o test test.c

EXAMPLE

test.c

```
#include<stdio.h>

int main()
{
        int a[]= {6,5,4,3,2};
        for (int i= 0; i < sizeof(a) / sizeof(int); i++) {
                if(a[i] > a[i+1])
                        continue;
                puts("Errore");
                break;
        }
        return 0;
}
```

```
MacBook-Francesco:Programmi francescosantini$ gcc -g -o test test.c
MacBook-Francesco:Programmi francescosantini$ ./test
Errore
```

EXAMPLE

You can start by entering the command **list**, or just its initial **l** for short, to list a few lines of source code of the program you are debugging.

```
MacBook-Francesco:Programmml francescosantini$ gdb test
(gdb) l
1           #include<stdio.h>
2
3           int main()
4           {
5                       int a[]= {6,5,4,3,2};
6
7                       for (int i= 0; i < sizeof(a) / sizeof(int); i++) {
8
9                               if(a[i] < a[i+1])
10                                      continue;
(gdb)
```

10 lines by default, l again

HOW TO WORK WITH IT

- Before you instruct GDB to run the program, you should tell it where you want it to stop.
- You can do this by setting a *breakpoint*.
- When the debugger reaches the breakpoint, it interrupts the execution of your program, giving you an opportunity to examine the program's state at that point.
- Furthermore, once the program has been interrupted at a breakpoint, you can continue execution line by line, observing the state of program objects as you go.
- To set a breakpoint, enter the command **break**, or **b** for short. Breakpoints are usually set at a specific line of source code or at the beginning of a function.

EXAMPLE

```
(gdb) l
1           #include<stdio.h>
2
3           int main()
4           {
5                       int a[]= {6,5,4,3,2};
6
7                       for (int i= 0; i < sizeof(a) / sizeof(int); i++) {
8
9                                   if(a[i] > a[i+1])
10                                              continue;
(gdb) b 7
Breakpoint 1 at 0x100000ecc: file test.c, line 7.
(gdb) r
Starting program: /Users/francescosantini/Desktop/ProgrammI/test

Breakpoint 1, main () at test.c:7
7                       for (int i= 0; i < sizeof(a) / sizeof(int); i++) {
```

EXAMPLE

- Upon reaching the breakpoint, the debugger interrupts the execution of the program and displays the line containing the next statement to be executed.
- For this purpose, GDB provides the commands **next**, or **n**, and **step**, or **s**. The next and step commands behave differently if the next line to be executed contains a function call.
- The **next** command executes the next line, including all function calls, and interrupts the program again at the following line.
- The **step** command, on the other hand, executes a jump to the function called in the line, and interrupts the program again at the first statement in the function body.

PRINT VARIABLES

- At this point we can check to see whether the values of the variables are correct. We can do this using the **print** command (**p** for short), which displays the value of a given expression:

$*number* is a GDB variable that the debugger creates.

```
(gdb) s
9                                   if(a[i] > a[i+1])
(gdb) p a[i]
$1 = 6
(gdb) p a[i+1]
$2 = 5
(gdb) p a[i+2]
$3 = 4
```

EXAMPLE

```
Breakpoint 1, main () at test.c:7
7                   for (int i= 0; i < sizeof(a)/ sizeof(int); i++) {
(gdb) n
9                           if(a[i] > a[i+1])
(gdb) n
10                                  continue;
(gdb) n
7                   for (int i= 0; i < sizeof(a)/ sizeof(int); i++) {
(gdb) n
9                           if(a[i] > a[i+1])
(gdb) n
10                                  continue;
(gdb) n
7                   for (int i= 0; i < sizeof(a)/ sizeof(int); i++) {
(gdb) n
9                           if(a[i] > a[i+1])
(gdb) n
10                                  continue;
(gdb) n
7                   for (int i= 0; i < sizeof(a)/ sizeof(int); i++) {
(gdb) n
9                           if(a[i] > a[i+1])
(gdb) n
10                                  continue;
```

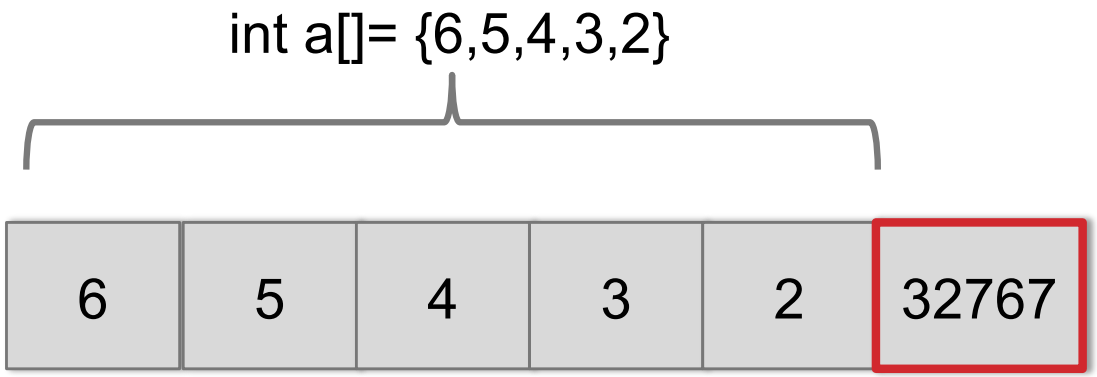
EXAMPLE

```
(gdb) n
7
(gdb) n
9
(gdb) n
12
(gdb) p a[i]
$2 = 2
(gdb) p a[i+1]
$3 = 32767
(gdb) p i
$4 = 4
```

```
for (int i= 0; i < sizeof(a) / sizeof(int); i++) {

        if(a[i] > a[i+1])

        puts("Errore");
```



LAST COMMANDS

- The command **continue**, abbreviated **c**, lets program execution continue until it reaches the next breakpoint or the end of the program.
- To stop gdb, enter the command **quit** or **q**.
- **info** breakpoints, **delete** breakpoints

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000100000f01 in main at test.c:10
            breakpoint already hit 3 times
(gdb) d 1
(gdb) c
Continuing.
Errore
```

SU LIBRO

- Sezioni 1.2, 1.4, 1.9.2-1.9.7
- Capitolo 14
- Appendice G