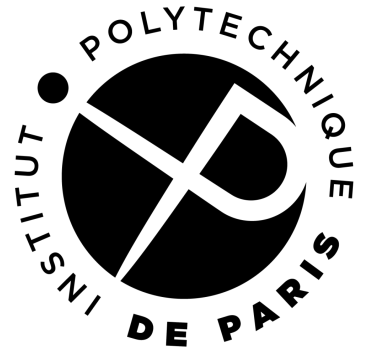


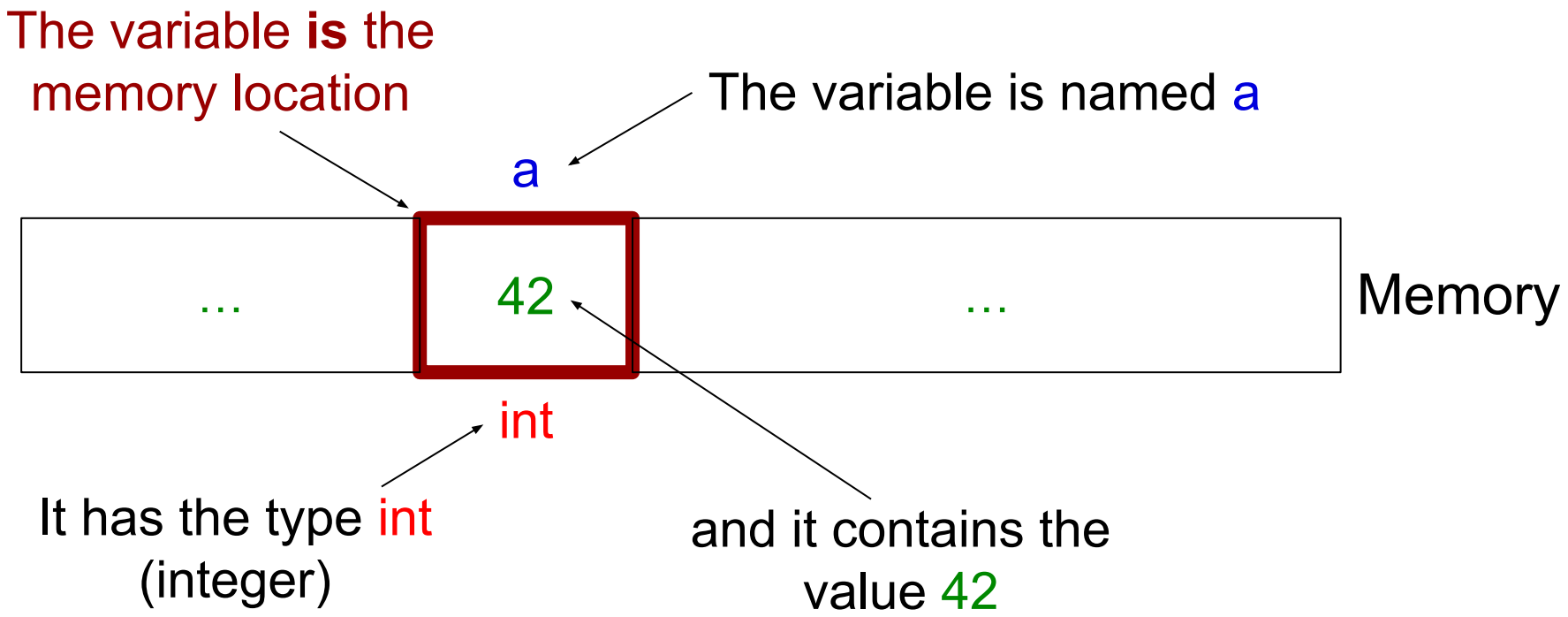
Functions, variables and call frames

Bachelor of Science - École polytechnique

gael.thomas@inria.fr

Remainder: the variable

- A variable **is a memory location**
  - that has name, a type and a value



Key concepts

- Function declaration and invocation
- Global variables, local variables and arguments
- Call frames
- Execution of a process

Function declaration

- A function is a group of instructions that creates a macro-instruction
  - Allows for code reuse
  - Can take arguments and return a result

- A method has:
  - A name: uniquely identify the function
  - A list of input parameters in the form of `type symbol`
  - A return type

```
int my_function(int a, int b) {
   ...
}
```

Function implementation

- A method has a body
  - A sequence of statements
  - Delimited by `{` and `}`
  - that ends with `return` statements (`return;` for `void` functions)

```
int my_function(int a, int b) {
  if(a == 42) {
    return 5;
  } else {
    return 3;
  }
  // or return a == 42 ? 5 : 3
}
```

Function invocation

- Usage
  - Invocation with `name_function(arg0, arg1...);`
  - When the function returns, the result replaces the invocation

```c
int add(int x, int y) {
  int z = x + y;
  return z;
}

int main(int argc, char* argv[]) {
  int a = 12;
  int res = add(a, 30); // the invocation is replaced by 42
  printf("%d\n", res);  // => 42
  return 0;
}
```

Declaration versus implementation

- You cannot call a function if the compiler does not know it
  - Problem in case of co-recursive calls (f calls g and g calls f)
  - In that case, you can declare a function without implementing it

```
int g(int n); // function declaration

int f(int n) {
  if(n == 1) {
    return g(n - 1);  // f can call g
  } else {
    return 1;
  }
}

int g(int n) {
  printf("call g: %d\n");
  f(n);
}
```

The main function (1/2)

- When the system starts a program, it creates a process
  - A program is just an executable file in the file system
  - While a process is a running instance of a program
  - Multiple processes can run the same program simultaneously.

- When the system starts a process
  - It loads its program in memory
  - Call its `main` function
  - Gives it two parameters
    - the number of arguments on the command line
    - the arguments themselves
      (the 0th argument is the path to the program)

The main function (2/2)

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

int main(int argc, char* argv[]) {
  for(int i=0; i<argc; i++) {
    printf("Arg[%d]: %s\n", i, argv[i]);
  }
  return 0;
}
```

cmdline.c

```
$ ./cmdline a b c
Arg[0]: ./cmdline
Arg[1]: a
Arg[2]: b
Arg[3]: c
```

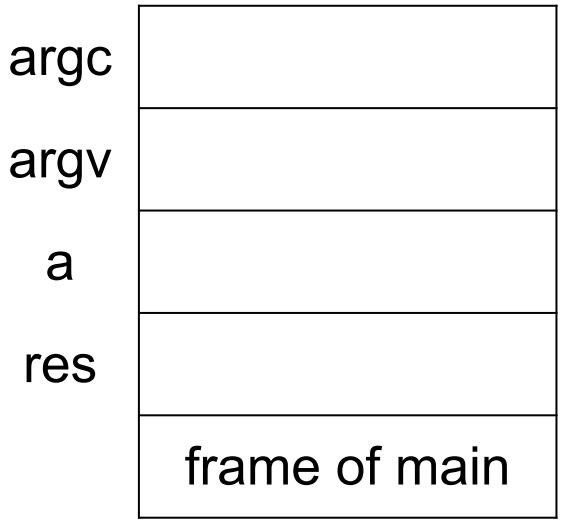
terminal

Local variables and parameters

- A variable defined in a function body is called a local variable
- Local variables and parameters exist only during an invocation
- For that, C manages what we call call frames
  - A call frame is a piece of memory that contains the local variables and parameters
  - Allocated when the function starts
  - Released when the function returns

Execution and call frames

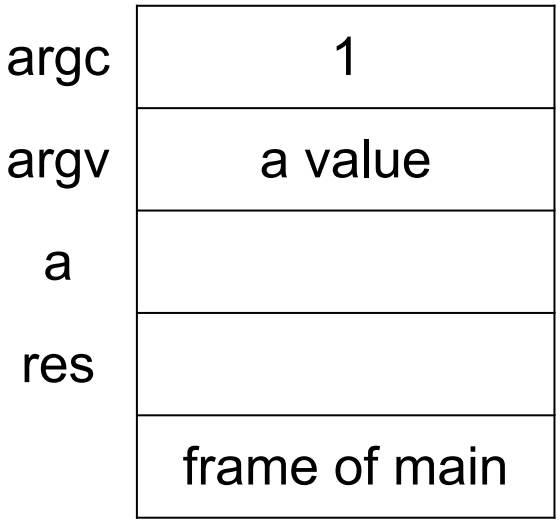
- When the program starts
  - Allocate a call frame for `main`




```
int add(int x, int y) {
  int z = x + y;
  return z;
}

int main(int argc, char* argv[]) {
  int a = 12;
  int res = add(a, 30);
  printf("%d\n", res);
  return 0;
}
```

Execution and call frames

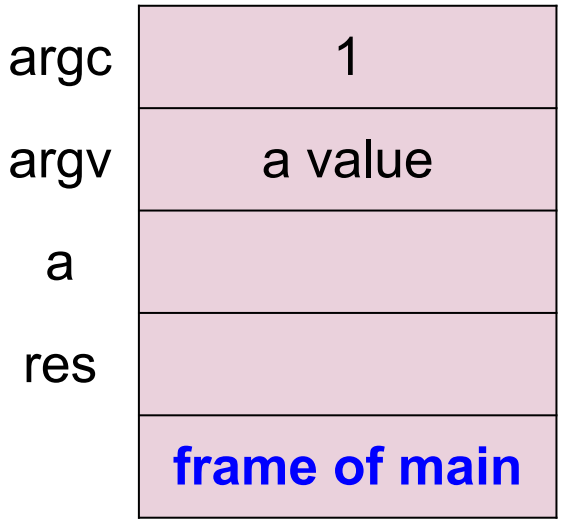
- When the program starts
  - Allocate a call frame for `main`
  - Install the arguments
    (argv is explained later in the course)




```
int add(int x, int y) {
  int z = x + y;
  return z;
}

int main(int argc, char* argv[]) {
  int a = 12;
  int res = add(a, 30);
  printf("%d\n", res);
  return 0;
}
```

Execution and call frames

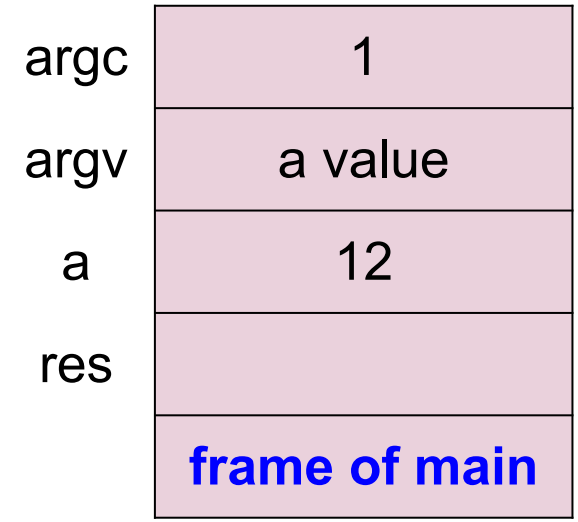
- When the program starts
  - Allocate a call frame for `main`
  - Install the arguments
  - Activate the frame of main
    (becomes the current frame)




```
int add(int x, int y) {
  int z = x + y;
  return z;
}

int main(int argc, char* argv[]) {
  int a = 12;
  int res = add(a, 30);
  printf("%d\n", res);
  return 0;
}
```

Execution and call frames

- In main
  - store 12 in a

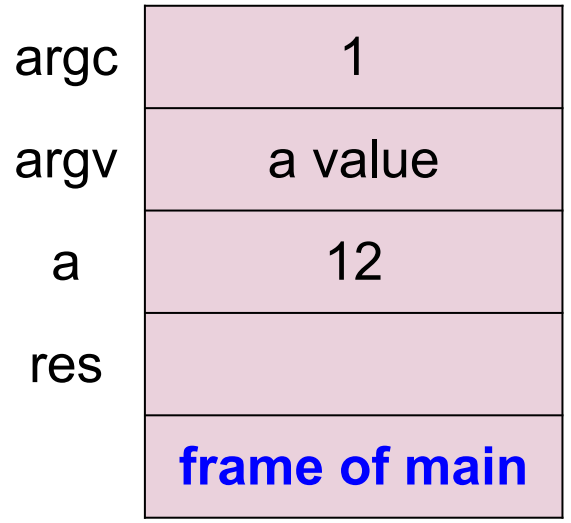


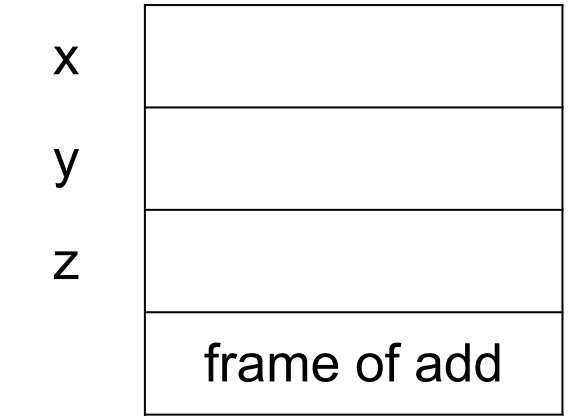

```
int add(int x, int y) {
  int z = x + y;
  return z;
}

int main(int argc, char* argv[]) {
  int a = 12;
  int res = add(a, 30);
  printf("%d\n", res);
  return 0;
}
```

Execution and call frames

- To call `add`
  - Allocate a frame for `add`

```
int add(int x, int y) {
  int z = x + y;
  return z;
}

int main(int argc, char* argv[]) {
  int a = 12;
  int res = add(a, 30);
  printf("%d\n", res);
  return 0;
}
```

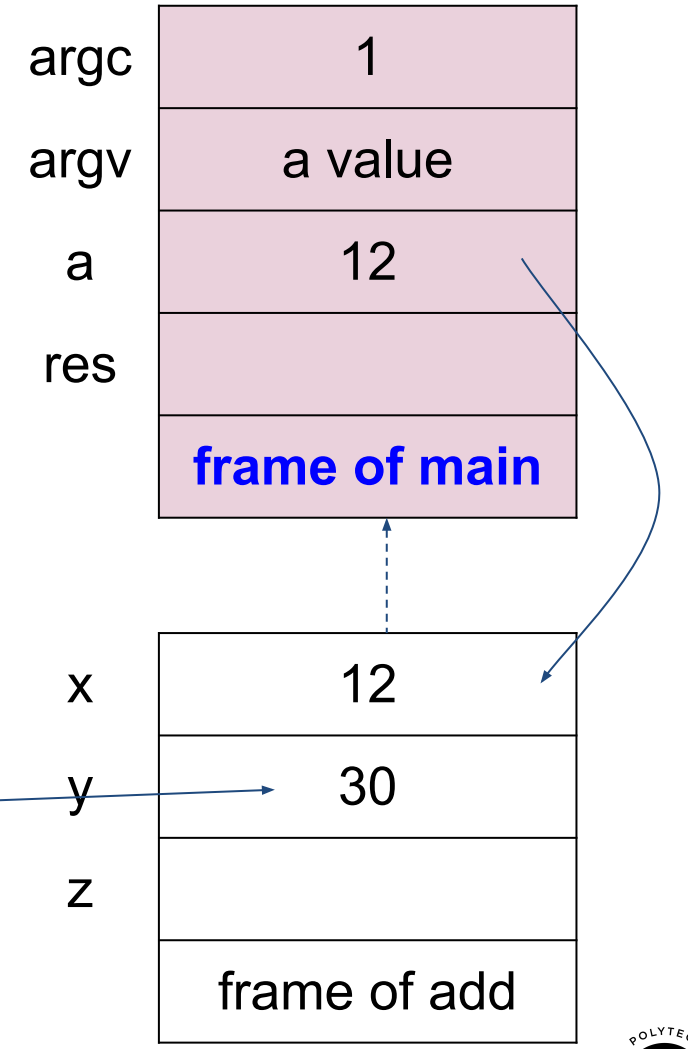




Execution and call frames

- To call `add`
  - Allocate a frame for `add`
  - Install the arguments

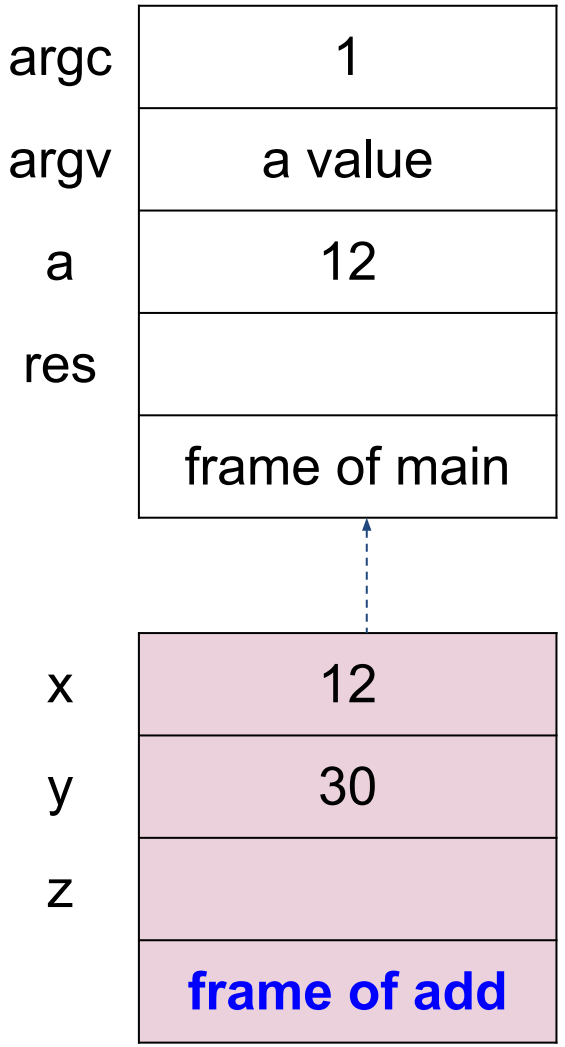
```
int add(int x, int y) {
  int z = x + y;
  return z;
}

int main(int argc, char* argv[]) {
  int a = 12;
  int res = add(a, 30);
  printf("%d\n", res);
  return 0;
}
```



Execution and call frames

- To call `add`
  - Allocate a frame for `add`
  - Install the arguments
  - Activate the frame

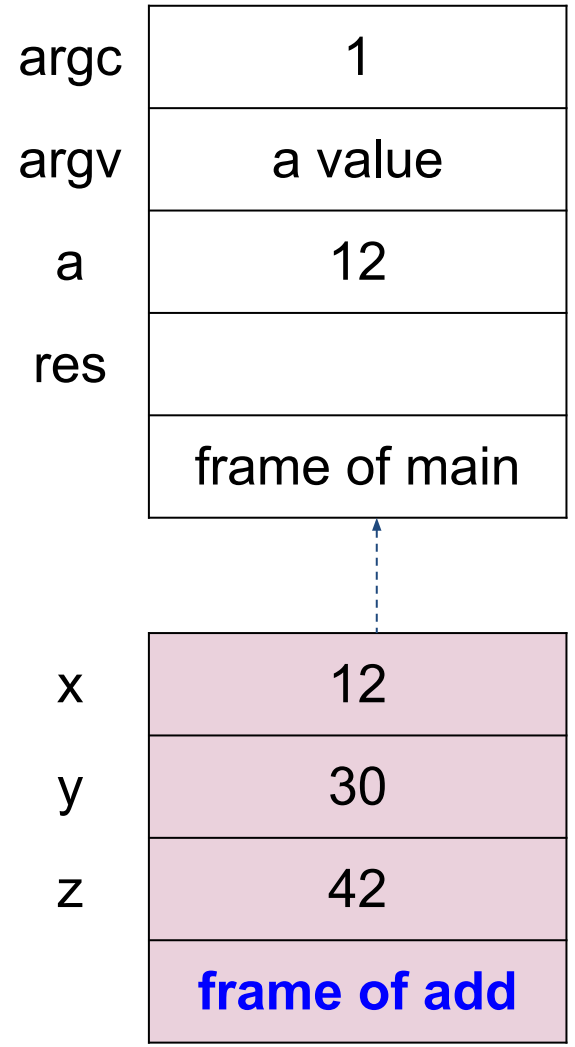
```
int add(int x, int y) {
  int z = x + y;
  return z;
}

int main(int argc, char* argv[]) {
  int a = 12;
  int res = add(a, 30);
  printf("%d\n", res);
  return 0;
}
```



Execution and call frames

- In `add`
  - Store `x + y` in `z`

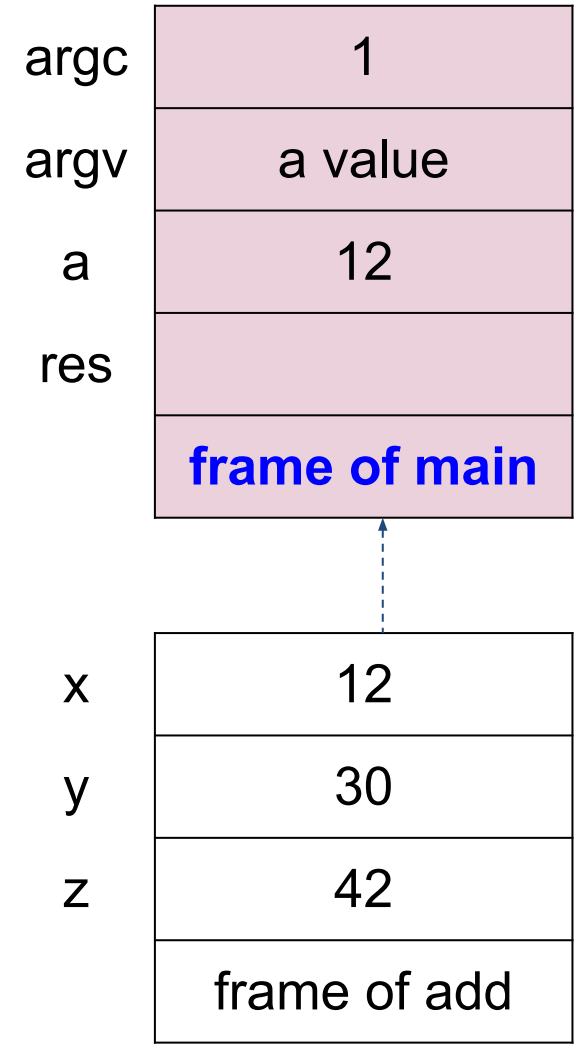
```
int add(int x, int y) {
  int z = x + y;
  return z;
}

int main(int argc, char* argv[]) {
  int a = 12;
  int res = add(a, 30);
  printf("%d\n", res);
  return 0;
}
```



Execution and call frames

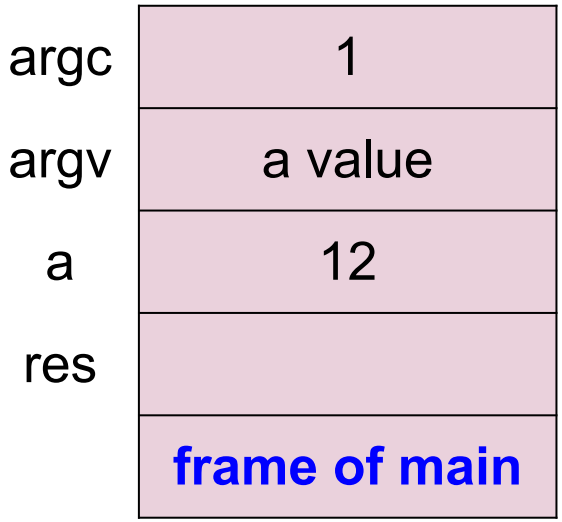
- In return
  - Activate the previous frame

```
int add(int x, int y) {
  int z = x + y;
  return z;
}

int main(int argc, char* argv[]) {
  int a = 12;
  int res = add(a, 30);
  printf("%d\n", res);
  return 0;
}
```



Execution and call frames

- In return
  - Activate the previous frame
  - Free the frame and return the result



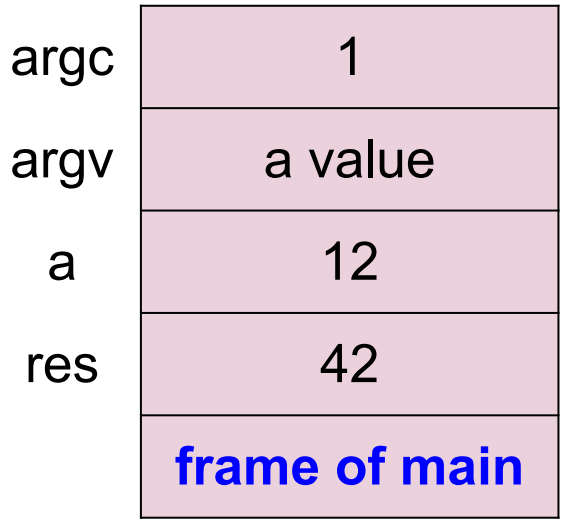

```
int add(int x, int y) {
  int z = x + y;
  return z;
}

int main(int argc, char* argv[]) {
  int a = 12;
  int res = add(a, 30);
  printf("%d\n", res);
  return 0;
}
```

42

Execution and call frames

- After the invocation
  - Set 42 in `res`

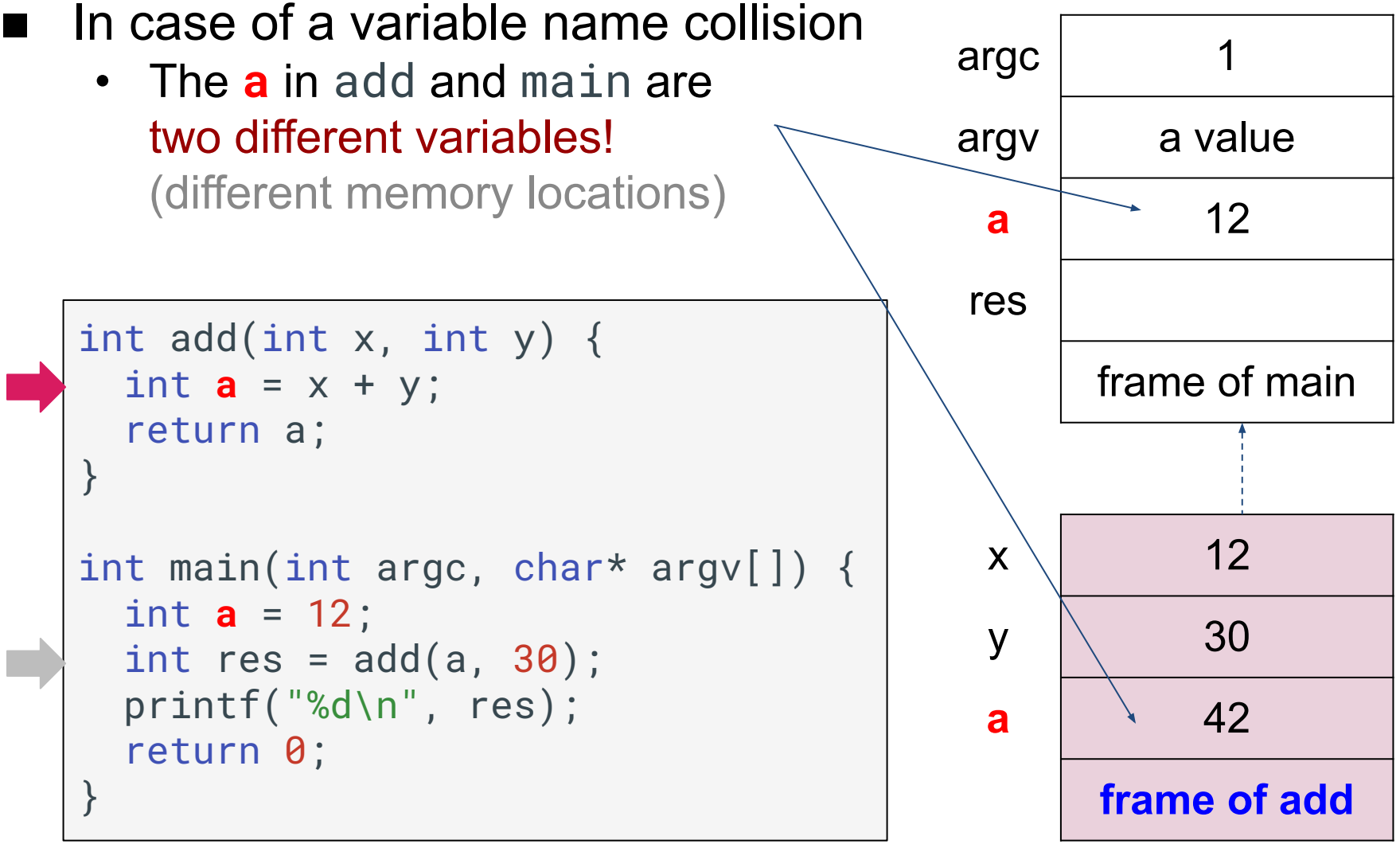
```
int add(int x, int y) {
  int z = x + y;
  return z;
}

int main(int argc, char* argv[]) {
  int a = 12;
  int res = add(a, 30);
  printf("%d\n", res);
  return 0;
}
```



Local variables and frames

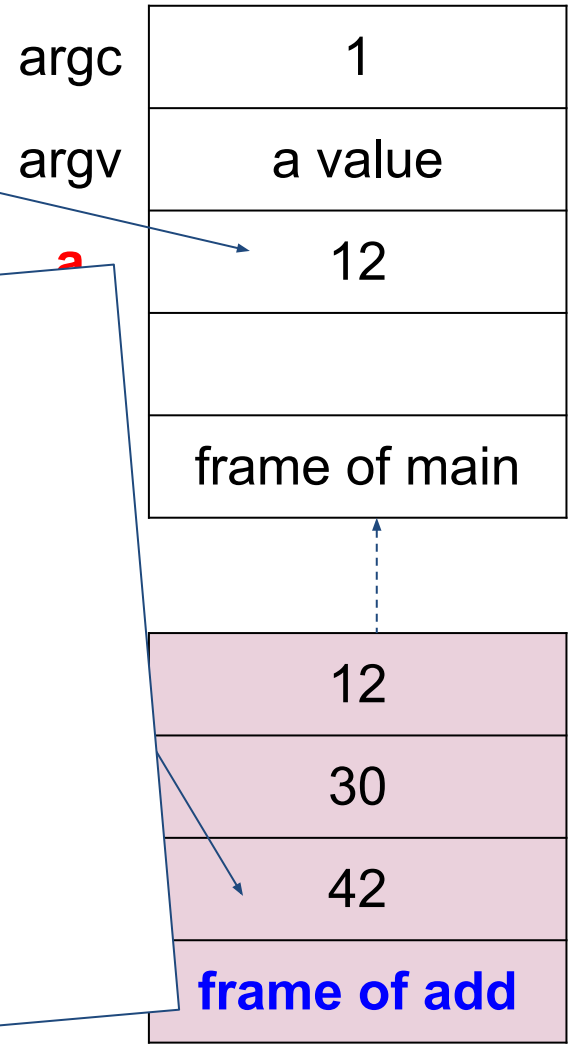
- In case of a variable name collision
  - The **a** in `add` and `main` are two different variables! (different memory locations)

```
int add(int x, int y) {
  int a = x + y;
  return a;
}

int main(int argc, char* argv[]) {
  int a = 12;
  int res = add(a, 30);
  printf("%d\n", res);
  return 0;
}
```



Local variables and frames

- In case of a variable name collision
  - The **a** in `add` and `main` are two different variables! (different memory locations)



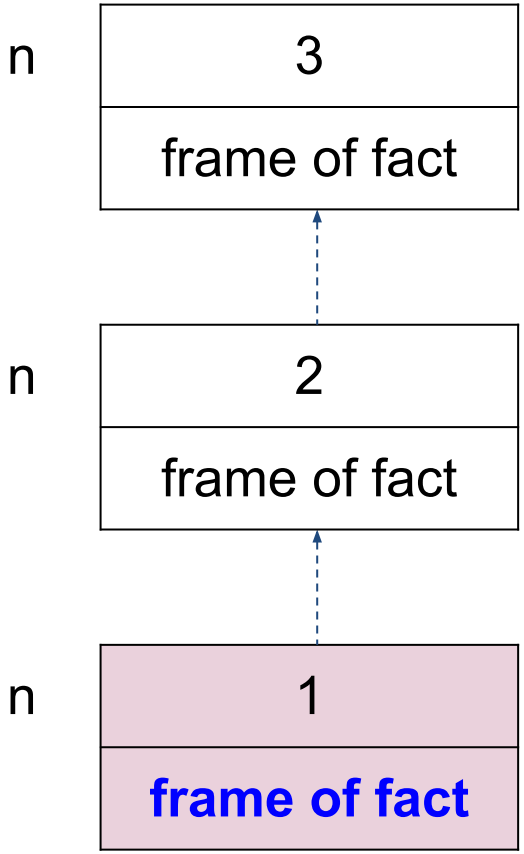
Modifying a in `add` does not modify a in `main`!

(in other words, a callee cannot change the value of a variable in a caller)

Call frames and recursive calls

- If a function calls itself
  - A new frame for each call
  - => new local variables for each call

```
int fact(int n) {
  if(n < 1)
    return 1;
  else
    return n * fact(n - 1);
}
// fact(3) calls fact(2),
// which calls fact(1)
```



Global variables

- A global variable is defined outside any function
  - Exists during the whole duration of the process
  - Can be read or written from any function
  - Can be hidden by a local variable or parameter with the same name (lexical scoping)

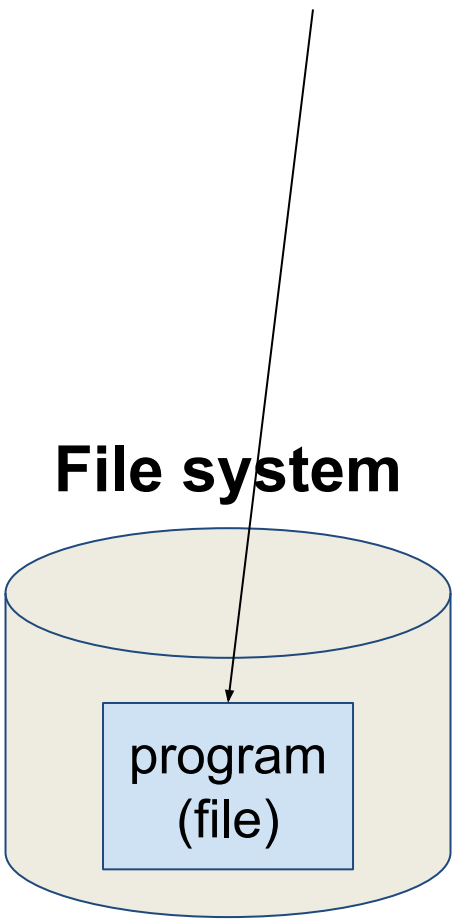
```
int a = 1, b = 2;

void test(int b) {
  a = 42;
  b = 42;
}

int main(int argc, char* argv[]) {
  test(666);
  printf("%d %d\n", a, b); // 42 2
}
```

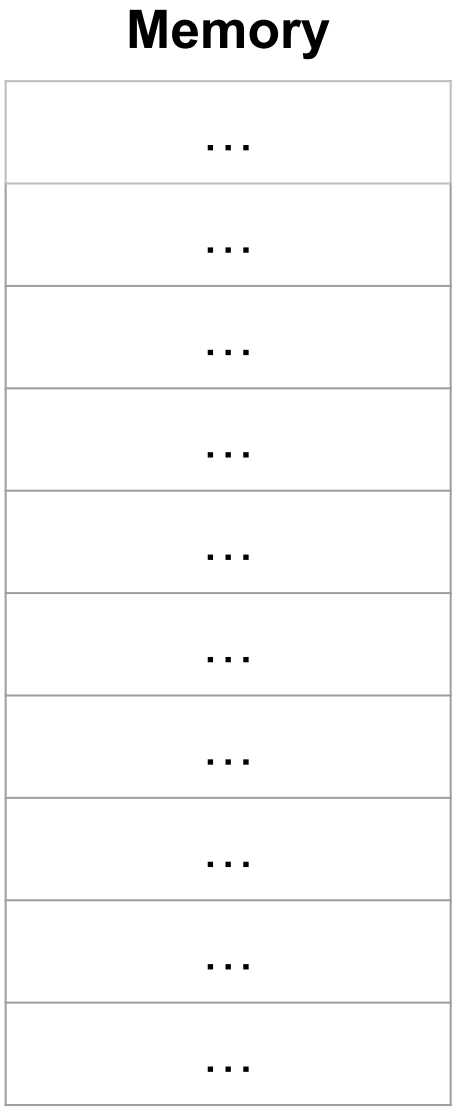
Modify the global variable

Modify the parameter, not the global variable

To execute a process

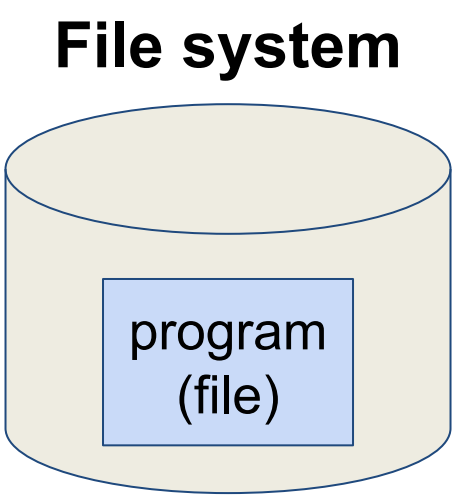
- Step 1: the operating system locates the program in the file system

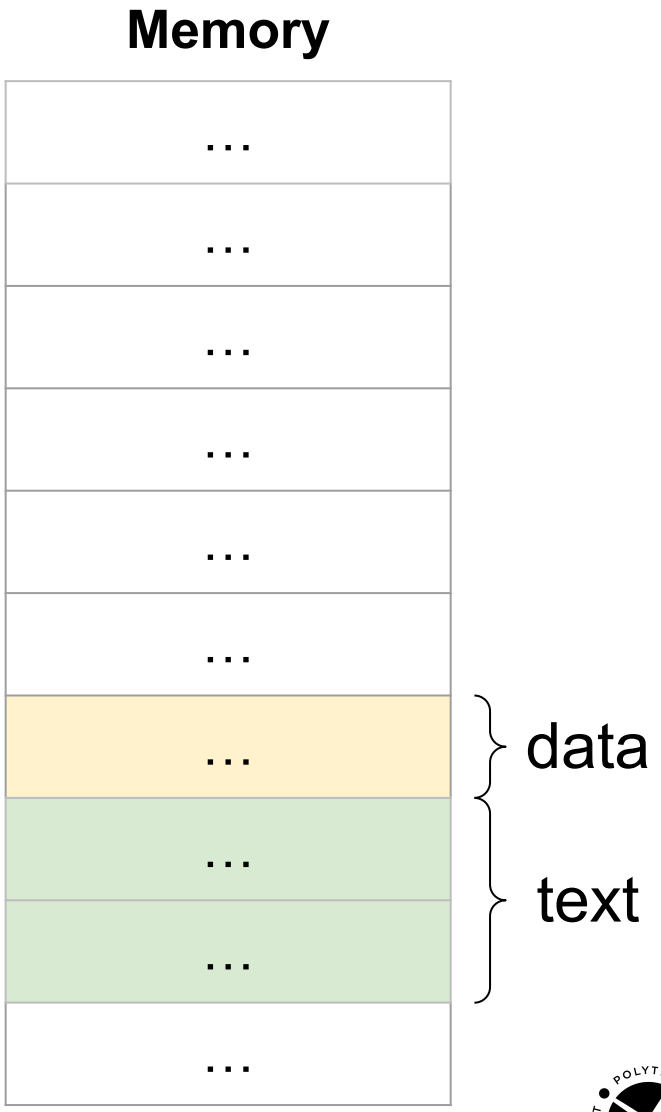




To execute a process

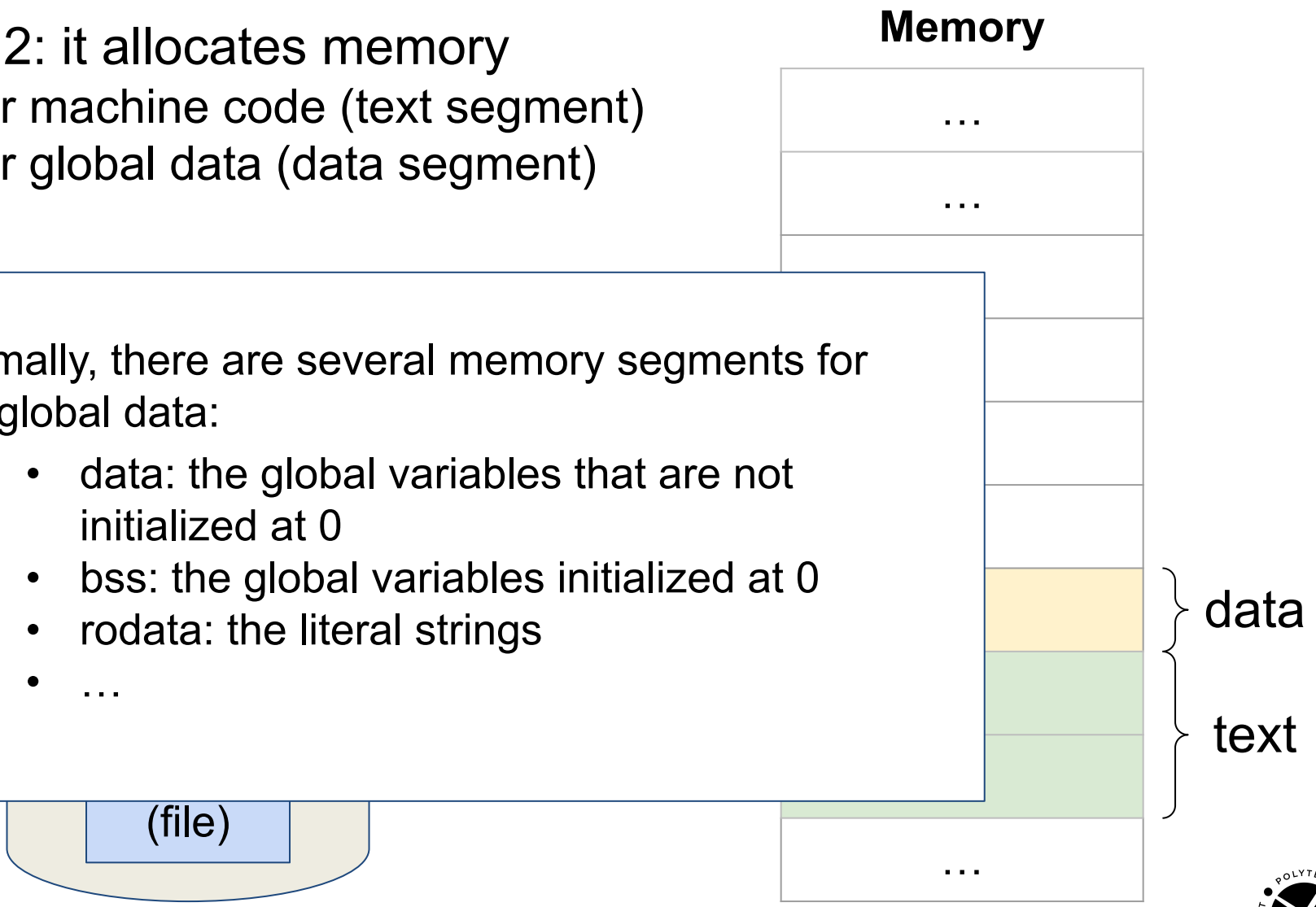
- Step 2: it allocates memory
  - for machine code (text segment)
  - for global data (data segment)





To execute a process

- Step 2: it allocates memory
  - for machine code (text segment)
  - for global data (data segment)



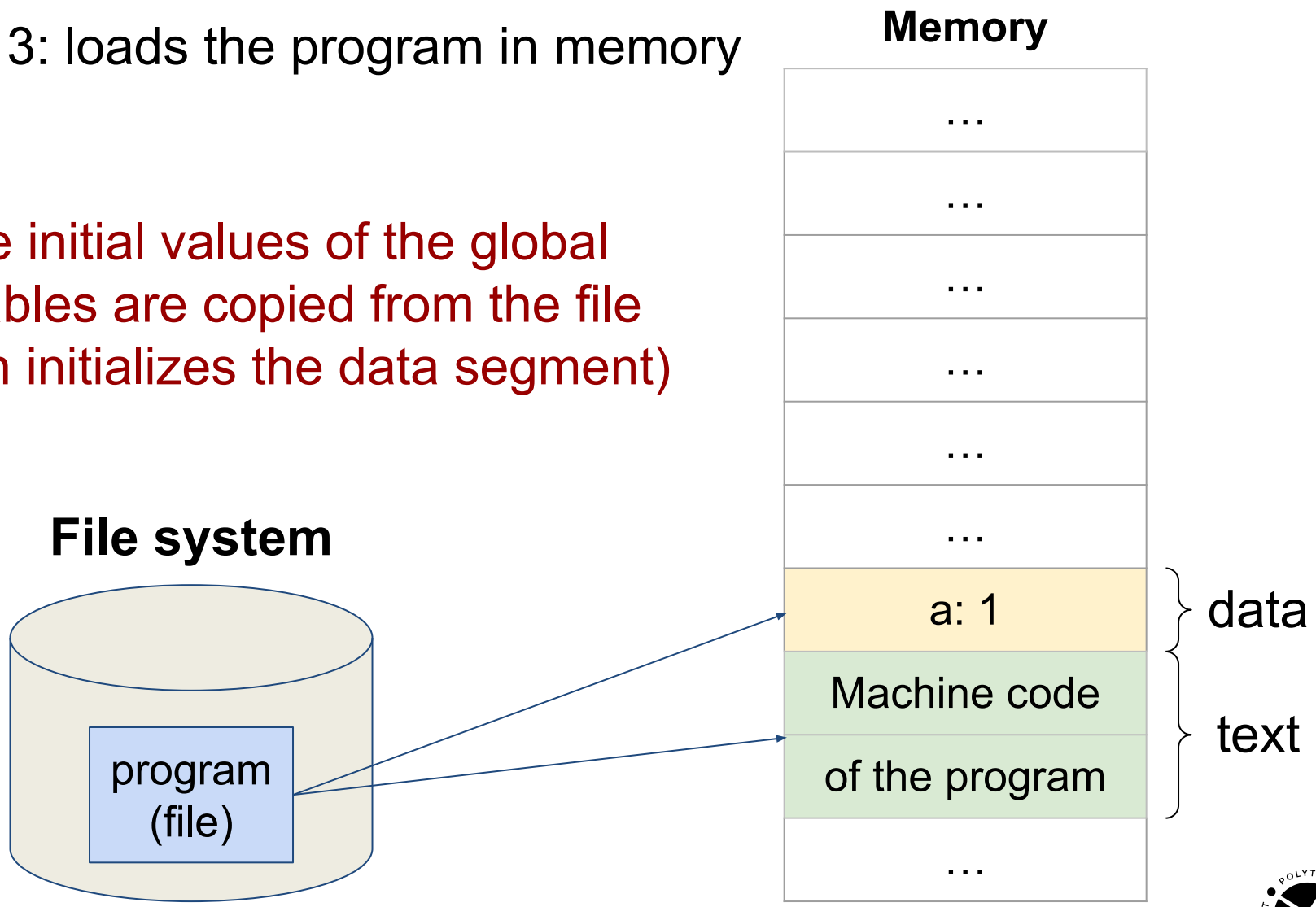
Formally, there are several memory segments for the global data:

- data: the global variables that are not initialized at 0
- bss: the global variables initialized at 0
- rodata: the literal strings
- …

To execute a process

- Step 3: loads the program in memory

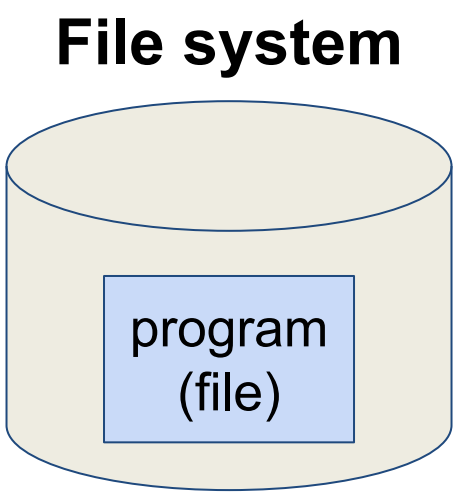
The initial values of the global variables are copied from the file (which initializes the data segment)

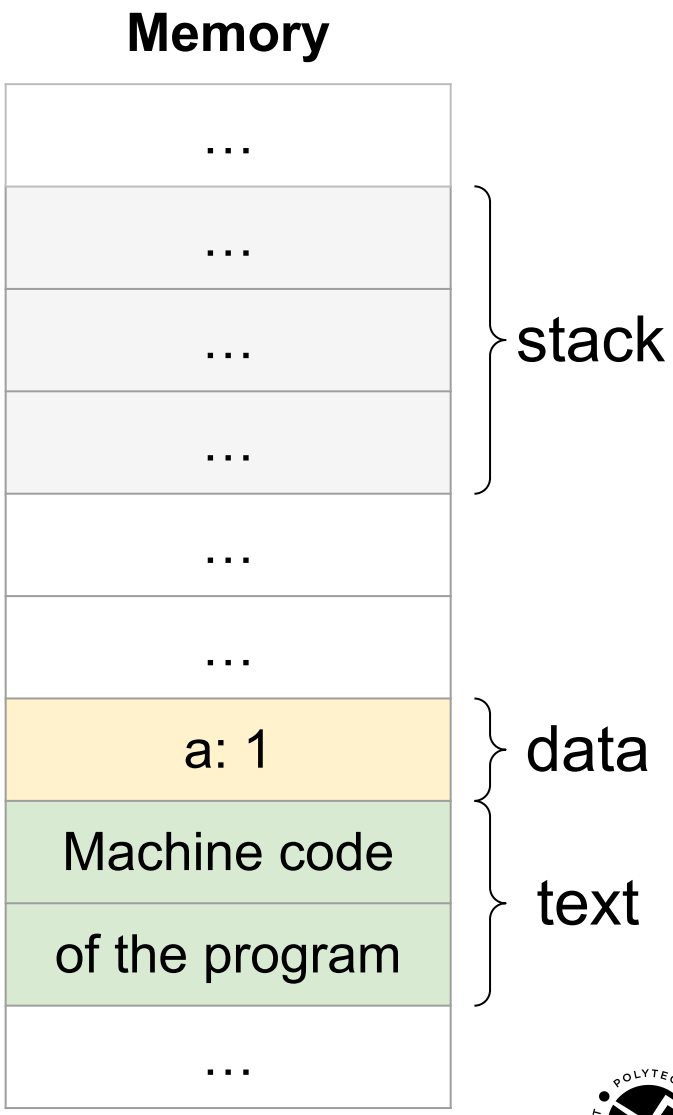


To execute a process

- Step 4: allocates memory for the call frames

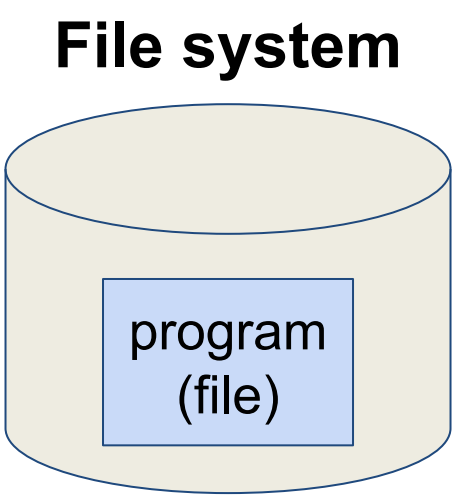
The memory zone that contains the call frames is called the stack segment

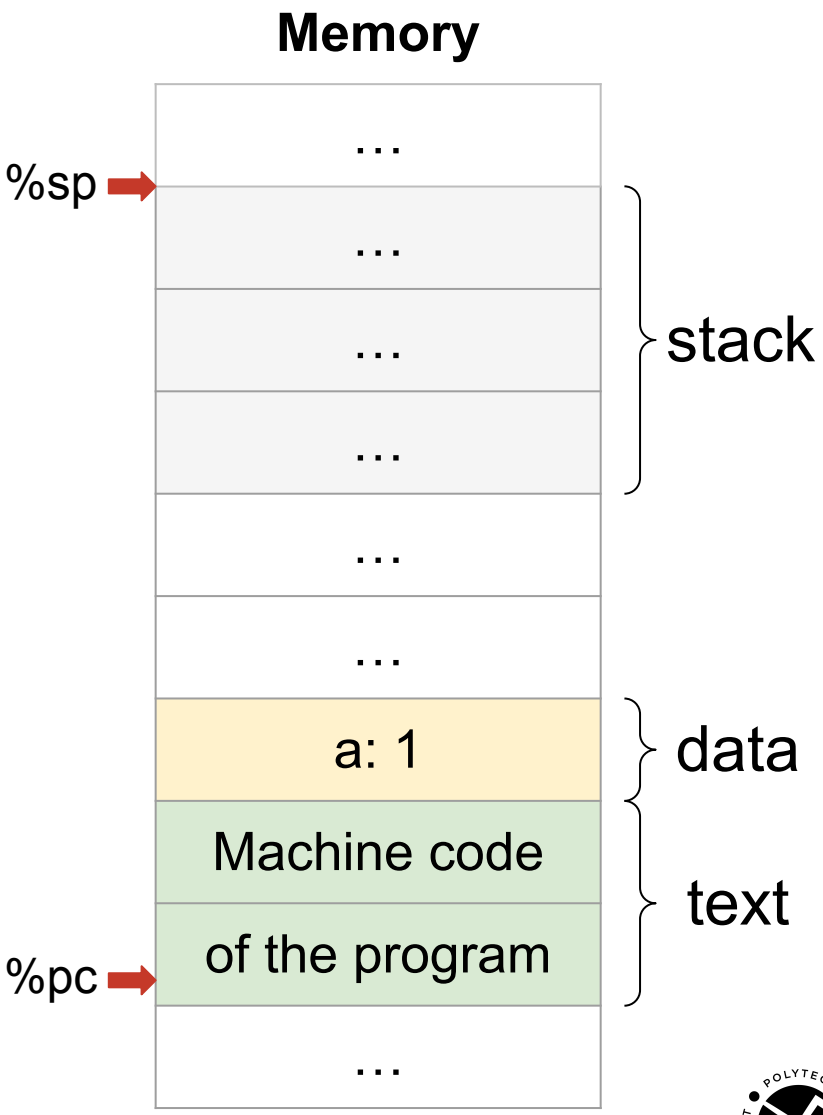




To execute a process

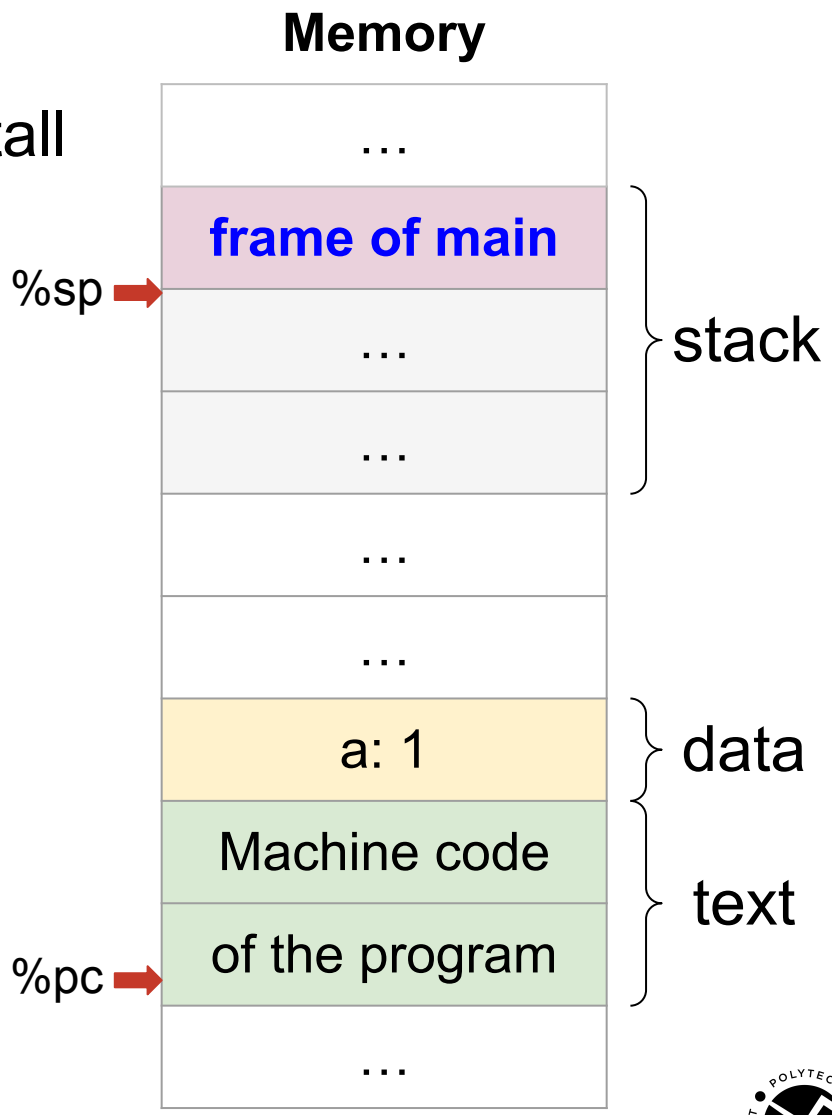
- Step 5: initialize two variables
  - The stack pointer (%sp) at the end of the stack
  - The program counter (%pc) at the beginning of the machine code of main

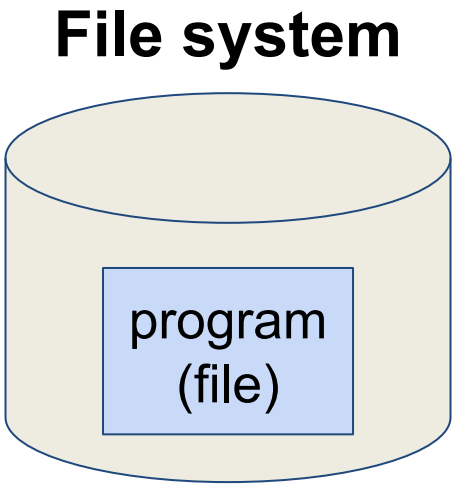




To execute a process

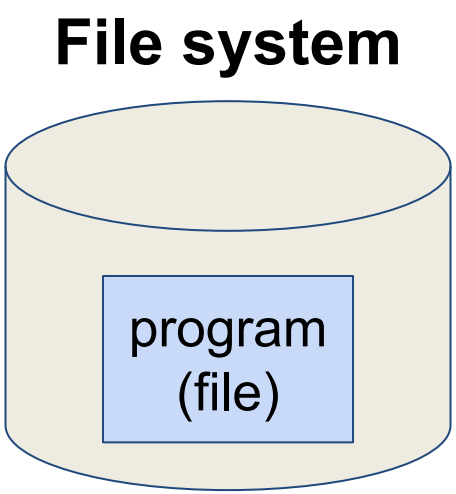
- Step 6: move %sp to make room for the call frame of main and install the arguments

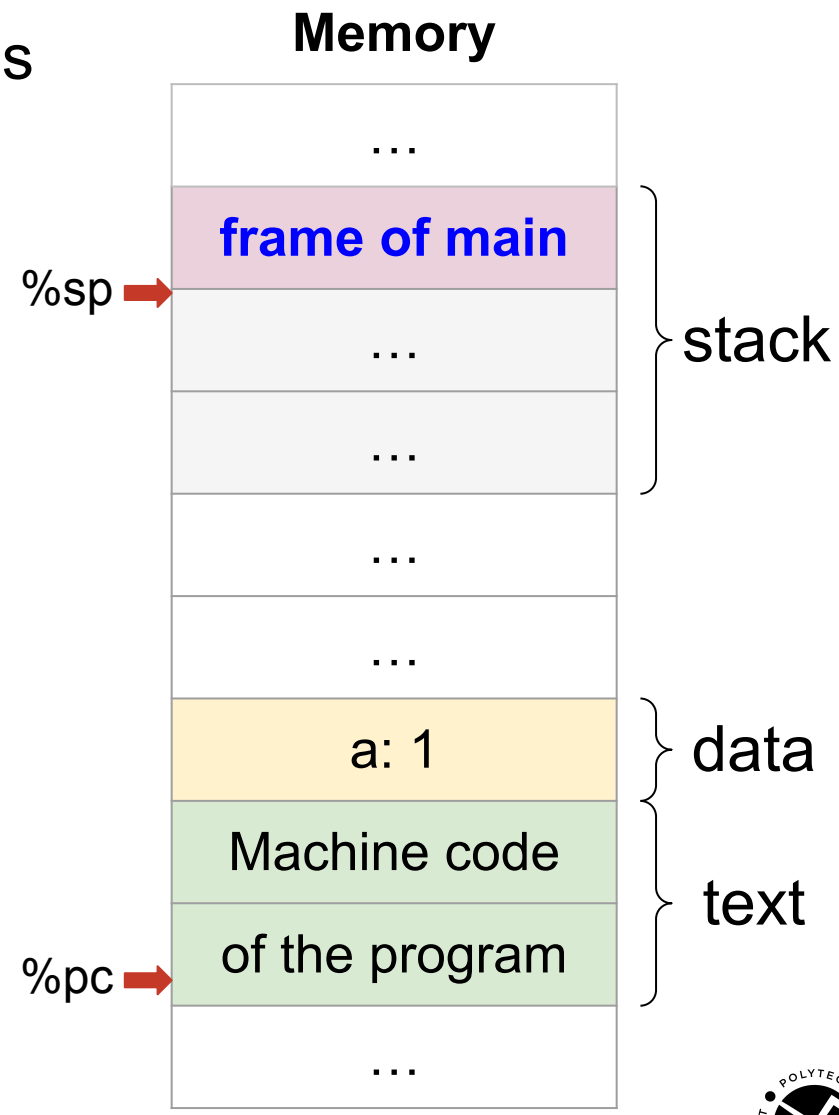




To execute a process

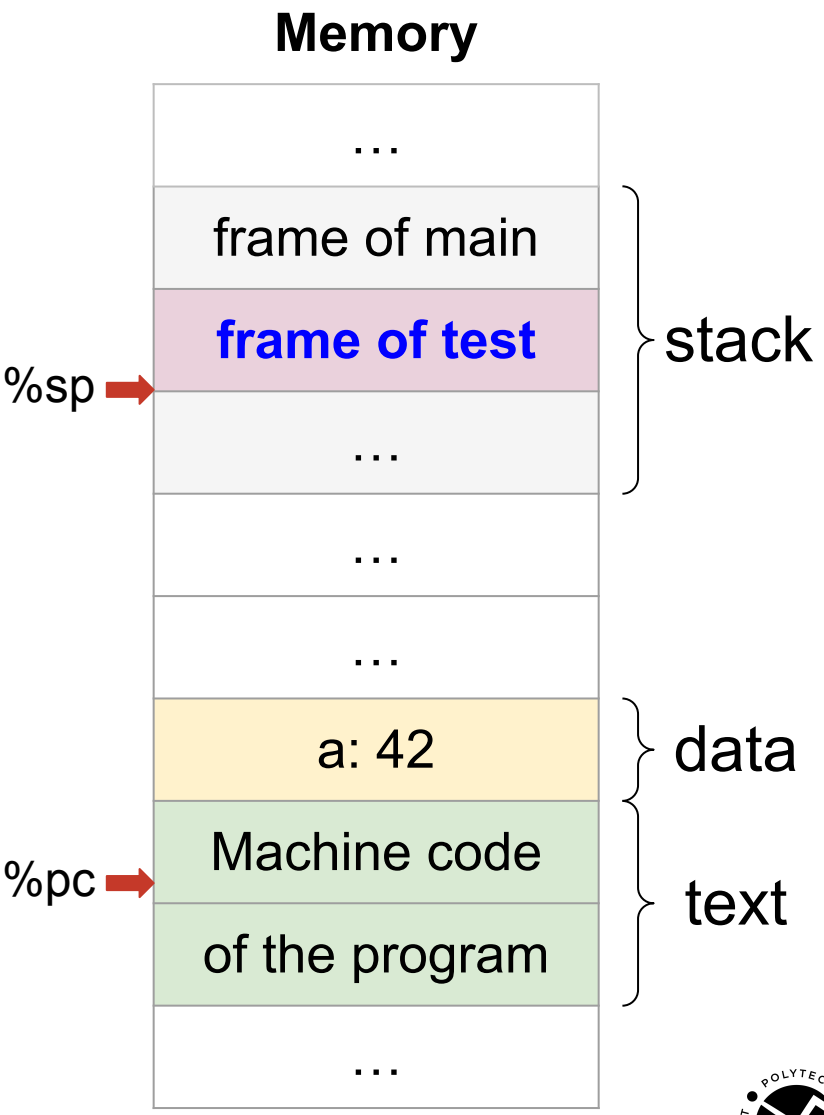
- Step 7: let the processor executes the code located at %pc

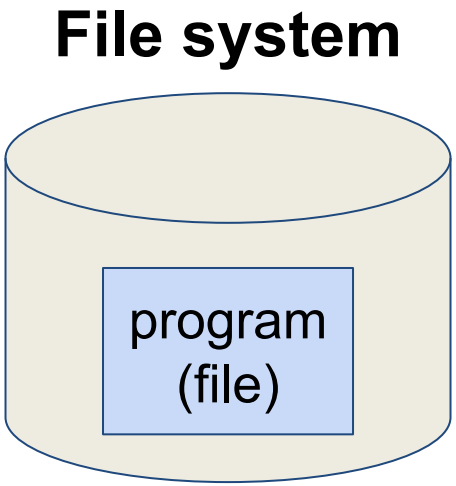




To execute a process

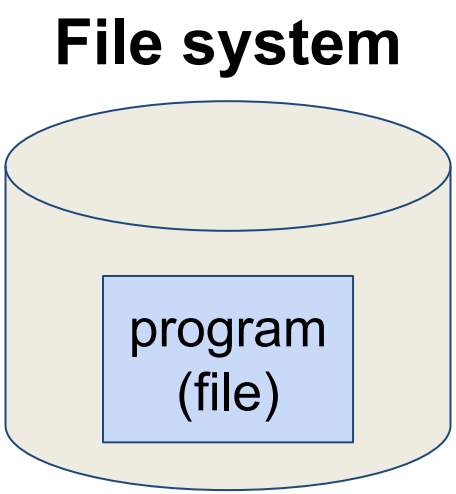
- Each time the code invokes a function, the machine code move accordingly %sp

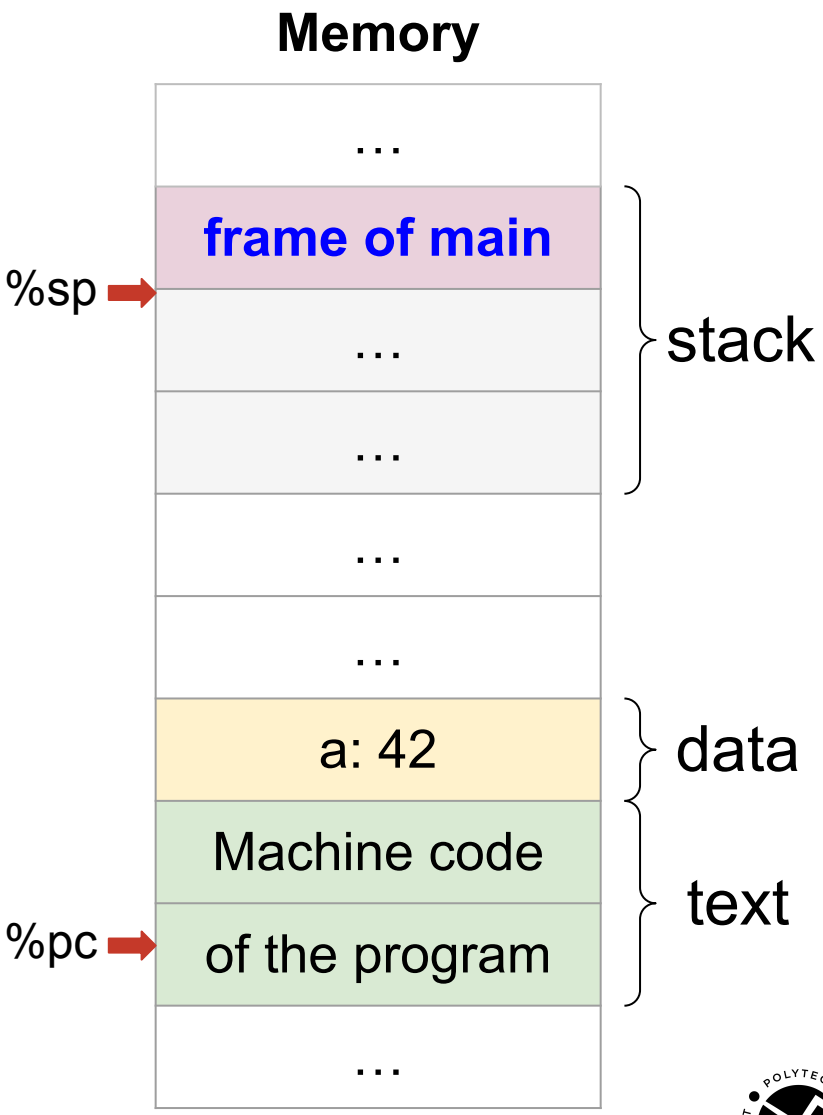




To execute a process

- Each time the code returns from a function, the machine code move accordingly %sp





Key concepts

- Function declaration and invocation
- Global variables, local variables and arguments
- Call frames
- Execution of a process