GNG1106
Fundamentals of Engineering Computation

Instructor: **Hitham Jleed**

University of Ottawa

Fall 2023 ~

Outline

Outline

- Modularization makes a program easier to code and maintain.
  - Modular program design: specific tasks are isolated and coded into separate modules or subprograms.
  - In C, a module or subprogram is called a function.
- Modular program design is advantageous since
  - it allows the abstraction of code during program design and development, and
  - it allows the re-use of code elsewhere in the program or in another program, hence avoiding repetition

- System built-in functions
  - `scanf()` and `printf()`
  - `fabs()`: computes the absolute value
  - `sine()`: computes the sine of an angle
  - `cosine()`: computes the cosine of an angle
  - `pow()`: computes the power of a number
  - ...
- `main()`: a special "sub-program", i.e., the main program.
- We can also create our own functions.

- Function in mathematics (say $f(x) = 3x + 5$):
  - it takes some input number
  - it generates some output number
- Function in C
  - it takes some input data
  - it generates (called "returns") some output data
  - and/or it does something
  - note: some of these components may be missing

Defining a Function in C

```
aType functionName(parameterList)
{
            declarations;
            commands;
}
```

- `parameterList` contains the declaration of one or a list of input variables that will receive input data
  - Individual declarations are separated by “,” (commas).
  - the list can be empty (or made as `void`) if the function takes no input
- `functionName` is the name of the function
- `aType` is the type of the value returned by the function.
- If `aType` is `void`, it means that the function returns nothing
- Variable declaration and commands are inside “{}”
- There is no “;” after }

A function with no argument and no return

```
void hello(void)
{
            printf("Hello World!\n");
}
```

A function with argument but without return

```
void prtmax(int a, int b)
{
      if (a > b)
          printf("%d is larger.\n", a);
      else
          printf("%d is larger.\n", b);
}
```

A function with both argument and return

```
int max(int a, int b)
{
      int larger=a;
      if (b > a)
            larger=b;
      return larger;
}
```

- A function that has non-void type must end with a statement `return blahBlah;`
- The returned value of the function need to have the type that is (or compatible with) the type of the function.
  - note: `larger` is int-typed; `larger` is returned; and the type of the function is int, matching the type of `larger`

Calling a Function

- Examples:
  - `hello();`
  - `prtmax(7, 9-6);`
  - `x=max(5, 9);`
- A call to a function can be made using the function name and (if the function is defined to accept inputs) a set of input values (in order of the input variables)
- When a function is called, the execution of the calling function is suspended and is resumed upon the completion of the function call.
- Calling a function having a non-void return type gives the return value of the function, e.g., `max(5,9)` gives value 9.

Returning from a Function

- When a function is executed without error, it returns the program control to the calling function.
- There are 3 ways of returning the program control from a function back to the caller:
  1. when reaching the end of the function as delimited by "`}`"
  2. by executing the command "`return;`"
  3. by executing the command "`return expression;`"
- Mechanisms 1 or 2 are used in a function that has no return type, i.e., having `void` type.
- Mechanism 3 must be used in a function that returns a non-void value; `expression` must evaluate to a value having the same type as the function's type.
- A function can only return ONE value (i.e., having ONE return type), but it may have many input values.

Scope and Lifetime

- Whenever a variable is declared and used in a program, two concepts are important: scope and lifetime.
- The scope of a variable is concerned with where the variable can be accessed.
- The lifetime of the variable is concerned with the duration for which the memory holding the variable will be reserved.

- The variables declared in the parameter list of a function and those declared inside the function (without using the "`static`" qualifier) are local[1]!
- This means:
  - The variables can only be accessed from within the function; they are invisible from outside the function.
  - The memory units reserved for the variables are released once the function returns; the variables "disappear" after the function returns.
- This rule applies to such variables declared in every function, including those declared in `main`.

[1] More precisely, these variables are local and "automatic", where the former refers to their scope and the latter refers to their lifetime.

Coding Demonstration

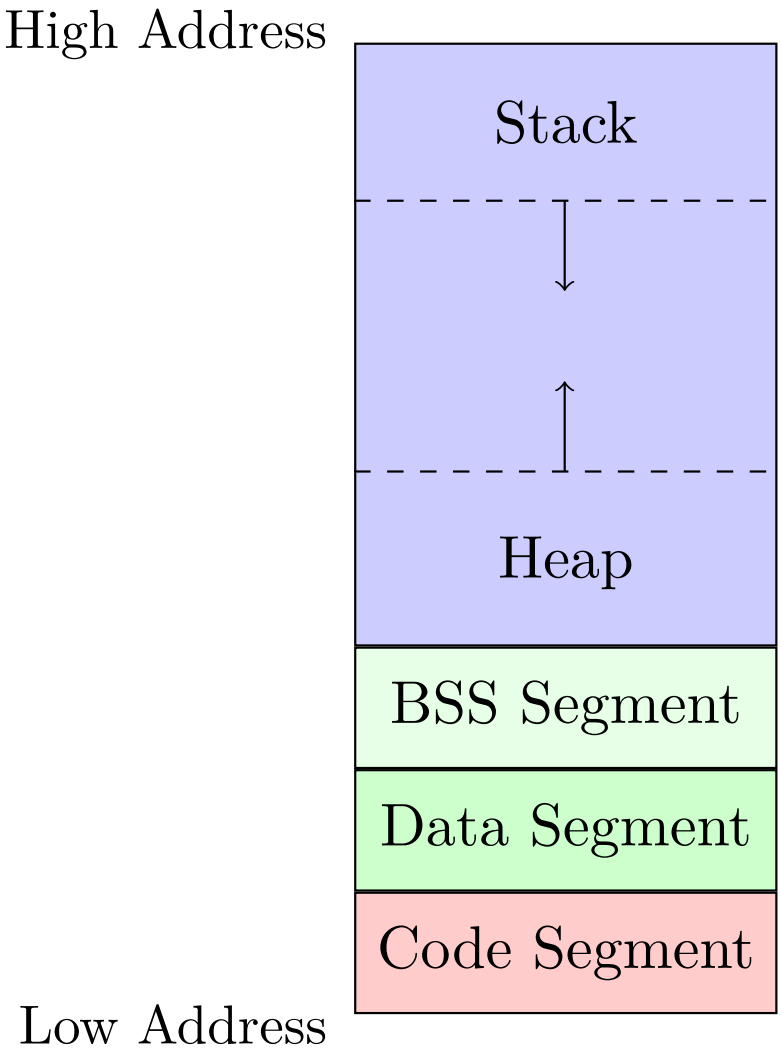
https://github.com/hjleed/GNG1106_Archive/tree/main/week6_code

Outline

Program Memory

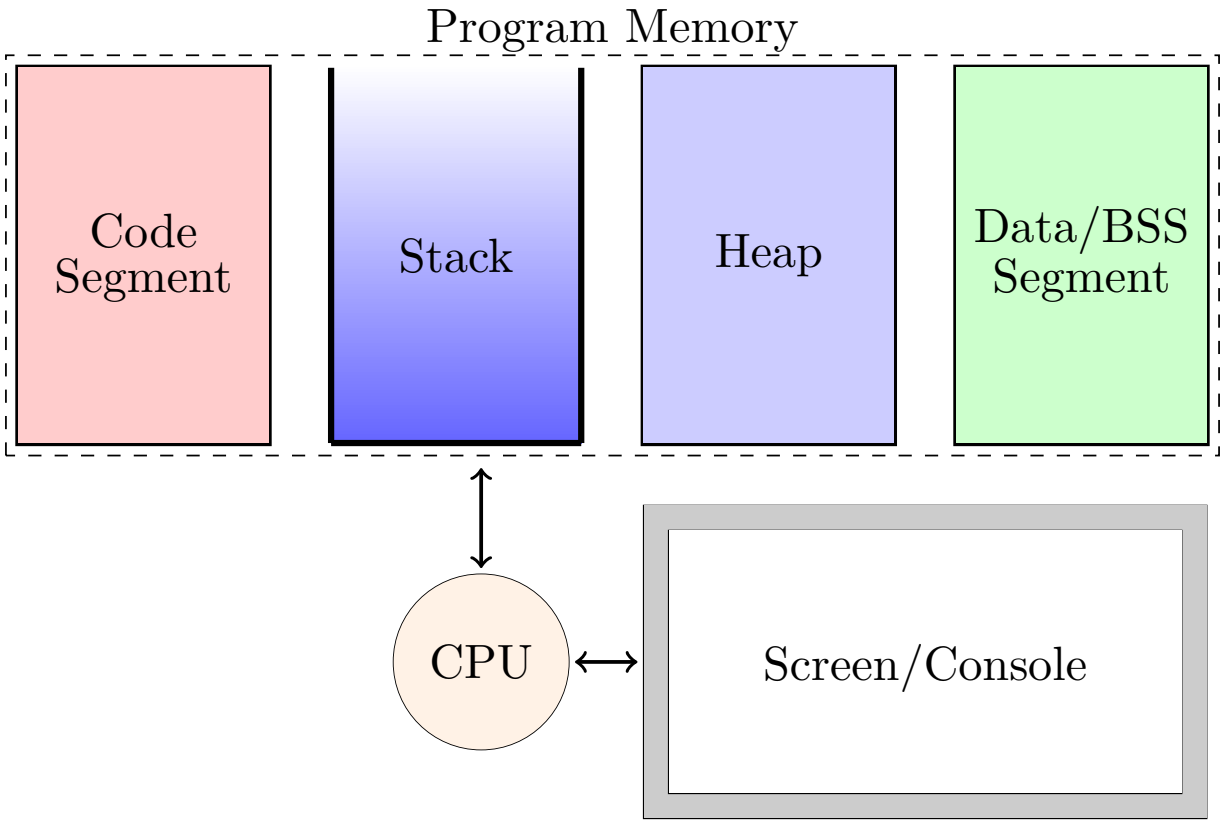
- The memory that a program uses is referred to as the program memory.
- Think the program memory as a long array of bytes, where each bytes has an address and every two neighbouring bytes have consecutive addresses.
- Such a memory structure is virtual, not physical.
- The operating systems in modern computers have taken care of this virtualization and hid from us the physical reality.
- As programmers, we will take this virtual structure as real as the physical reality!

Typical Layout of Program Memory



- Code segment stores the program.
- Data segment stores initialized global or static variables.
- BSS ("Block Started by Symbol", but never mind the name) segment stores uninitialized global or static variables.
- Heap is for dynamically allocated memory.
- Stack is for storing local variables during function calls.

"Programming Model" used in This Course



Tracing Program in Programming Model

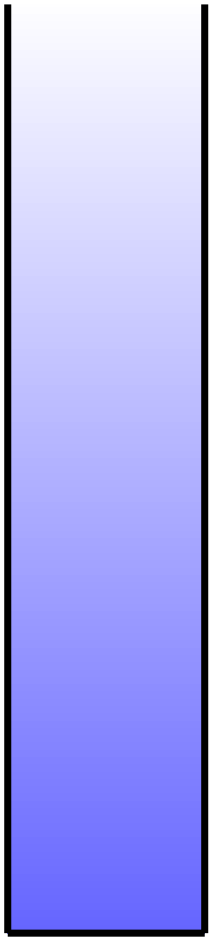
```
#include <stdio.h>
int sum(int a, int b)
{
    int c;
    c=a+b;
    return c;
}
int main()
{
    int x, y;
    printf("Enter 2 integers\n");
    scanf("%d%d", &x, &y);
    printf("The sum is %d\n", sum(x,
y));
    return 0;
}
```

Loading ...

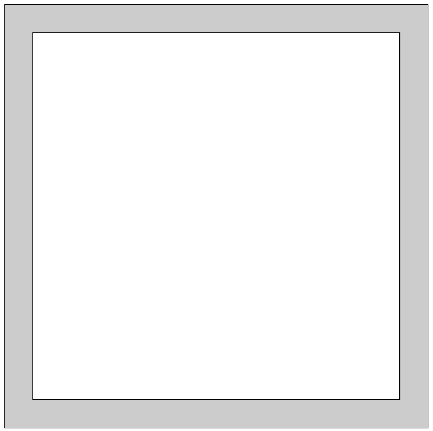
Code Segment

```
#include <stdio.h>
int sum(int a, int b)
{
   int c;
   c=a+b;
   return c;
}
int main()
{
   int x, y;
   printf("Enter 2 integers\n");
   scanf("%d%d", &x, &y);
   printf("The sum is %d\n",
sum(x, y));
   return 0;
}
```

Stack



Screen/Console

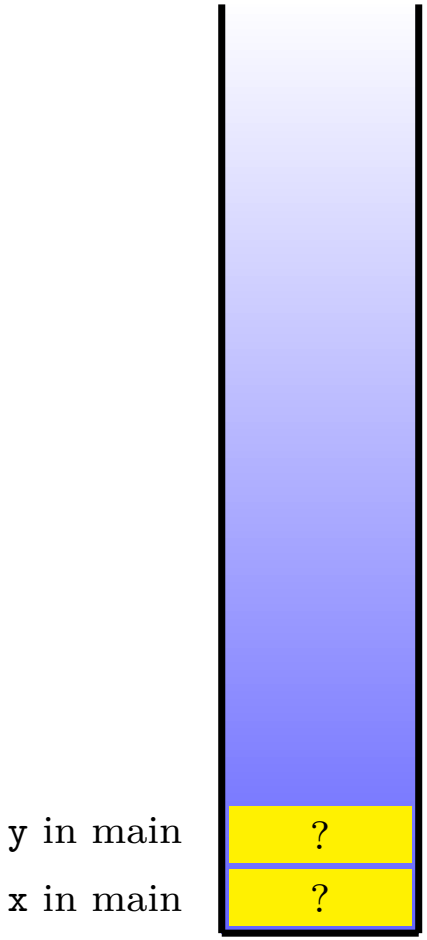


Executing 1 ...

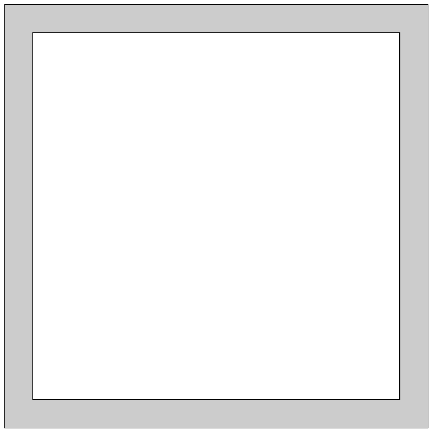
Code Segment

```
#include <stdio.h>
int sum(int a, int b)
{
   int c;
   c=a+b;
   return c;
}
int main()
{
   int x, y;
   printf("Enter 2 integers\n");
   scanf("%d%d", &x, &y);
   printf("The sum is %d\n",
sum(x, y));
   return 0;
}
```

Stack



Screen/Console

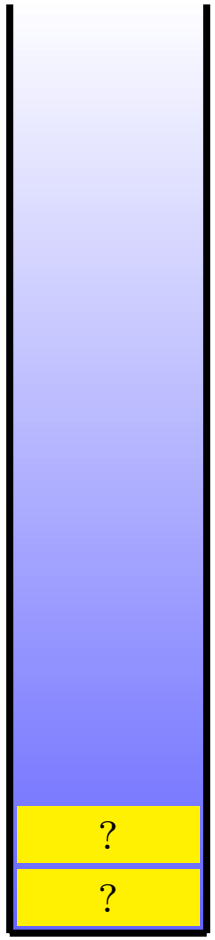


Executing 2 ...

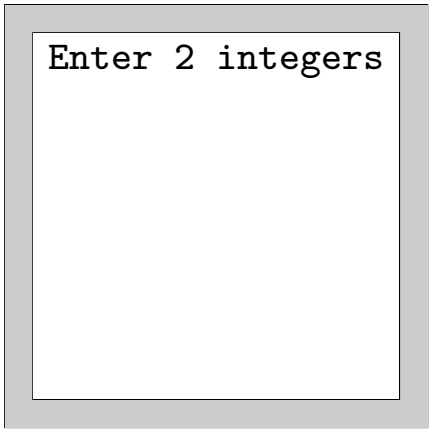
Code Segment

```
#include <stdio.h>
int sum(int a, int b)
{
   int c;
   c=a+b;
   return c;
}
int main()
{
   int x, y;
   printf("Enter 2 integers\n");
   scanf("%d%d", &x, &y);
   printf("The sum is %d\n",
sum(x, y));
   return 0;
}
```

Stack



Screen/Console

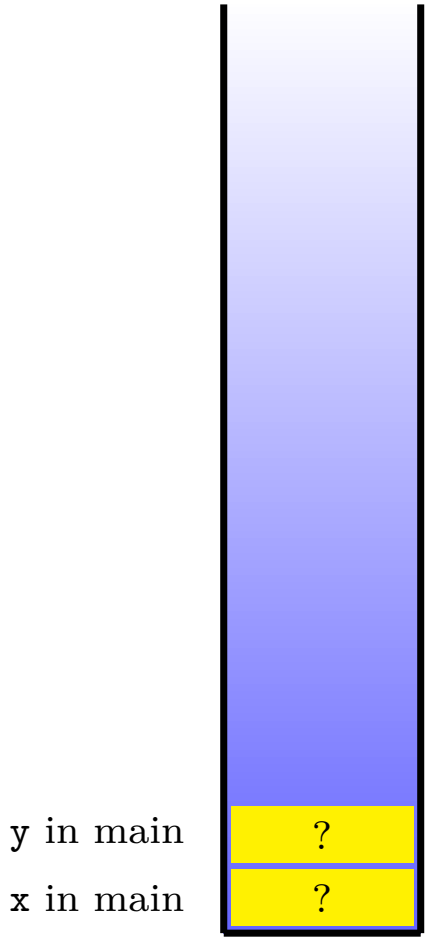


Executing 3 ...

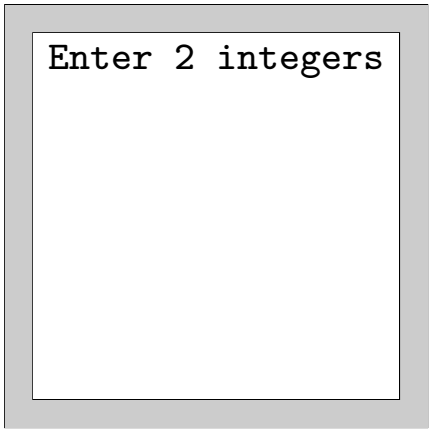
Code Segment

```
#include <stdio.h>
int sum(int a, int b)
{
   int c;
   c=a+b;
   return c;
}
int main()
{
   int x, y;
   printf("Enter 2 integers\n");
   scanf("%d%d", &x, &y);
   printf("The sum is %d\n",
sum(x, y));
   return 0;
}
```

Stack



Screen/Console

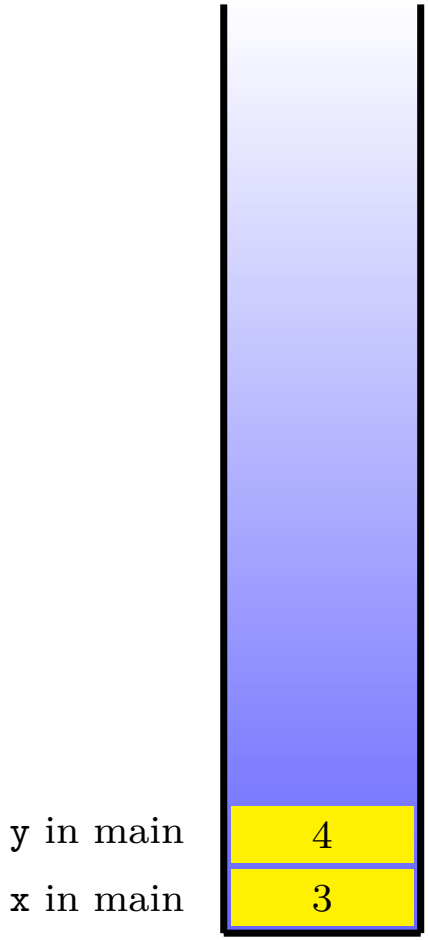


Executing 4 ...

Code Segment

```
#include <stdio.h>
int sum(int a, int b)
{
   int c;
   c=a+b;
   return c;
}
int main()
{
   int x, y;
   printf("Enter 2 integers\n");
   scanf("%d%d", &x, &y);
   printf("The sum is %d\n",
sum(x, y));
   return 0;
}
```

Stack



Screen/Console

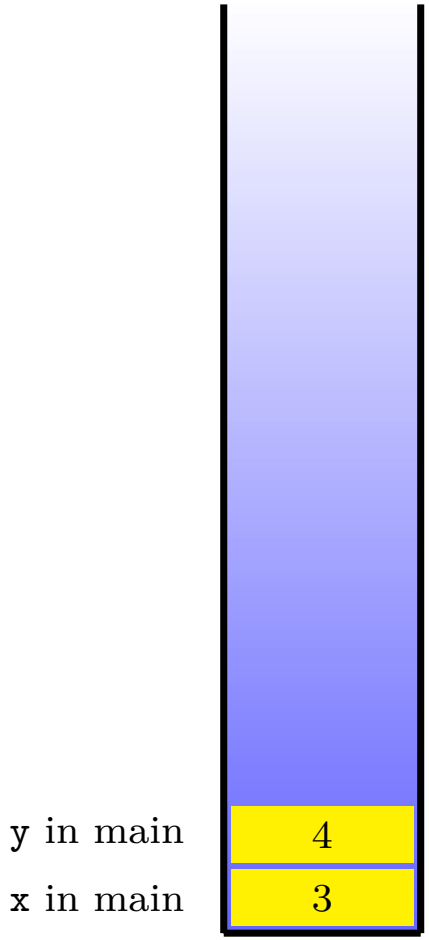
```
Enter 2 integers
3
4
```

Executing 5 ...

Code Segment

```
#include <stdio.h>
int sum(int a, int b)
{
   int c;
   c=a+b;
   return c;
}
int main()
{
   int x, y;
   printf("Enter 2 integers\n");
   scanf("%d%d", &x, &y);
   printf("The sum is %d\n",
sum(x, y));
   return 0;
}
```

Stack



Screen/Console

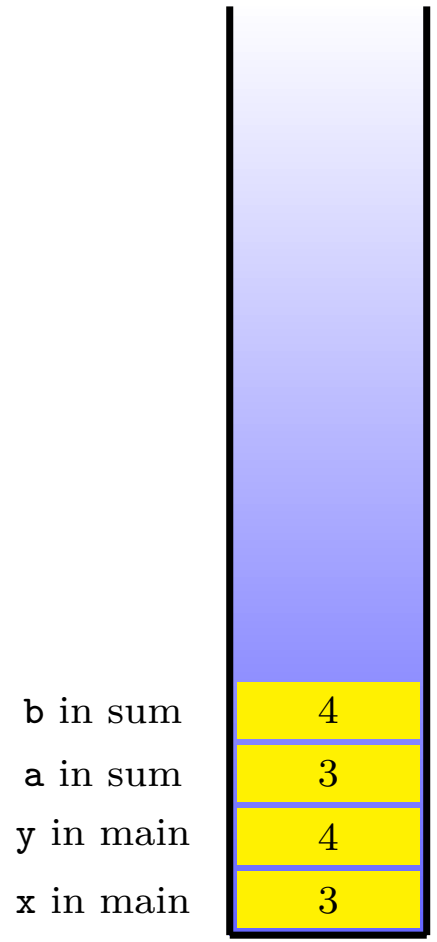
```
Enter 2 integers
3
4
```

Executing 6 ...

Code Segment

```
#include <stdio.h>
int sum(int a, int b)
{
   int c;
   c=a+b;
   return c;
}
int main()
{
   int x, y;
   printf("Enter 2 integers\n");
   scanf("%d%d", &x, &y);
   printf("The sum is %d\n",
sum(x, y));
   return 0;
}
```

Stack



Screen/Console

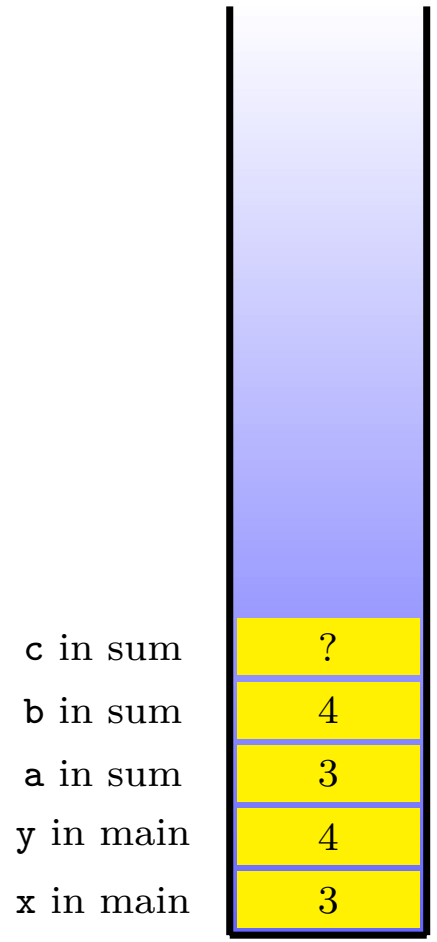
```
Enter 2 integers
3
4
```

Executing 7 ...

Code Segment

```
#include <stdio.h>
int sum(int a, int b)
{
   int c;
   c=a+b;
   return c;
}
int main()
{
   int x, y;
   printf("Enter 2 integers\n");
   scanf("%d%d", &x, &y);
   printf("The sum is %d\n",
sum(x, y));
   return 0;
}
```

Stack



Screen/Console

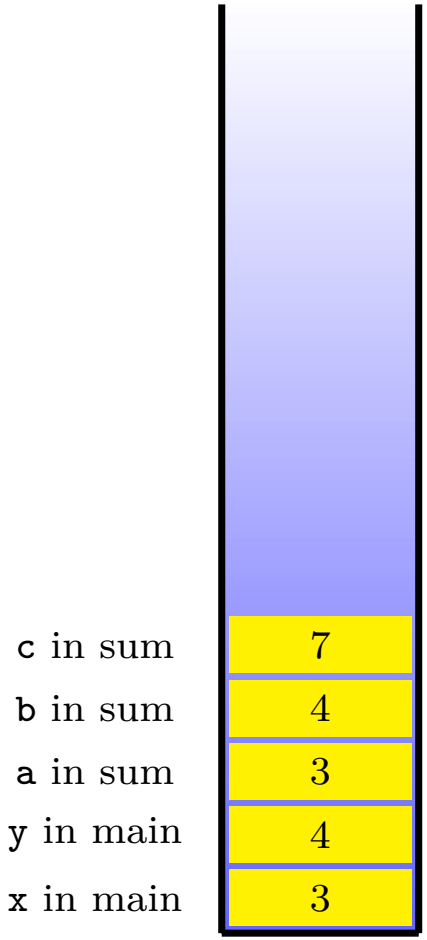
```
Enter 2 integers
3
4
```

Executing 8 ...

Code Segment

```
#include <stdio.h>
int sum(int a, int b)
{
   int c;
   c=a+b;
   return c;
}
int main()
{
   int x, y;
   printf("Enter 2 integers\n");
   scanf("%d%d", &x, &y);
   printf("The sum is %d\n",
sum(x, y));
   return 0;
}
```

Stack



Screen/Console

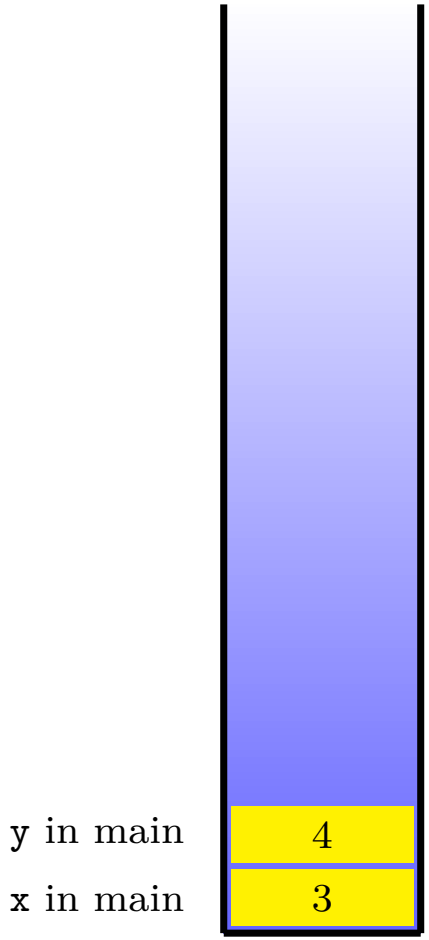
```
Enter 2 integers
3
4
```

Executing 9 ...

Code Segment

```
#include <stdio.h>
int sum(int a, int b)
{
   int c;
   c=a+b;
   return c;
}
int main()
{
   int x, y;
   printf("Enter 2 integers\n");
   scanf("%d%d", &x, &y);
   printf("The sum is %d\n",
sum(x, y));
   return 0;
}
```

Stack



Screen/Console

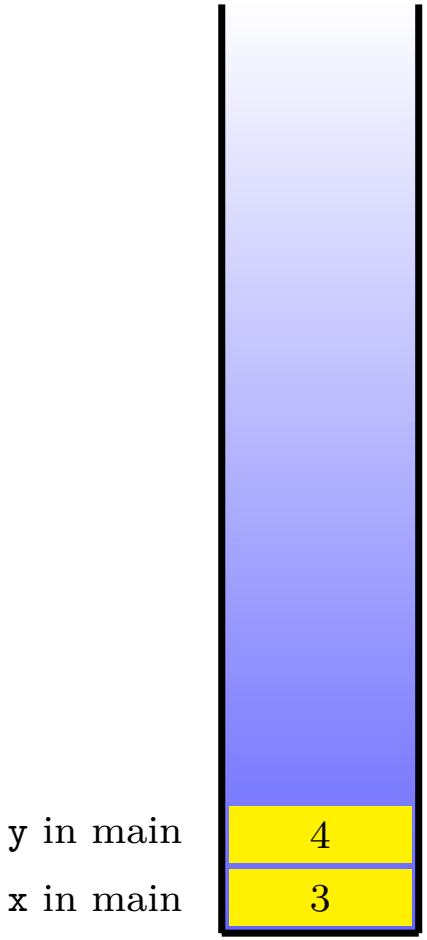
```
Enter 2 integers
3
4
```

Executing 10 ...

Code Segment

```
#include <stdio.h>
int sum(int a, int b)
{
   int c;
   c=a+b;
   return c;
}
int main()
{
   int x, y;
   printf("Enter 2 integers\n");
   scanf("%d%d", &x, &y);
   printf("The sum is %d\n",
sum(x, y));
   return 0;
}
```

Stack



Screen/Console

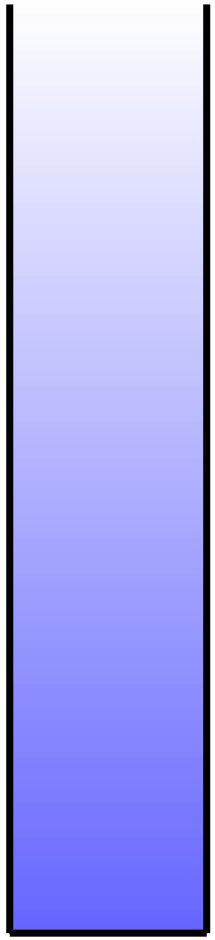
```
Enter 2 integers
3
4
The sum is 7
```

Executing 11 ...

Code Segment

```
#include <stdio.h>
int sum(int a, int b)
{
   int c;
   c=a+b;
   return c;
}
int main()
{
   int x, y;
   printf("Enter 2 integers\n");
   scanf("%d%d", &x, &y);
   printf("The sum is %d\n",
sum(x, y));
   return 0;
}
```

Stack



Screen/Console

```
Enter 2 integers
3
4
The sum is 7
```

- In this example, in fact the calling of the functions `printf` and `scanf` also involves memory allocation, which we do not trace in the programming model.
  - For example, in the final call of `printf`, the returned value 7 is used as an argument for `printf`. Necessarily, a local variable in function `printf` is used to stores the value 7. This variable also lives in the stack.
- When tracing a program, we ignore memory allocation in the calling of system built-in functions.
- The "programming model" is a MODEL after all.

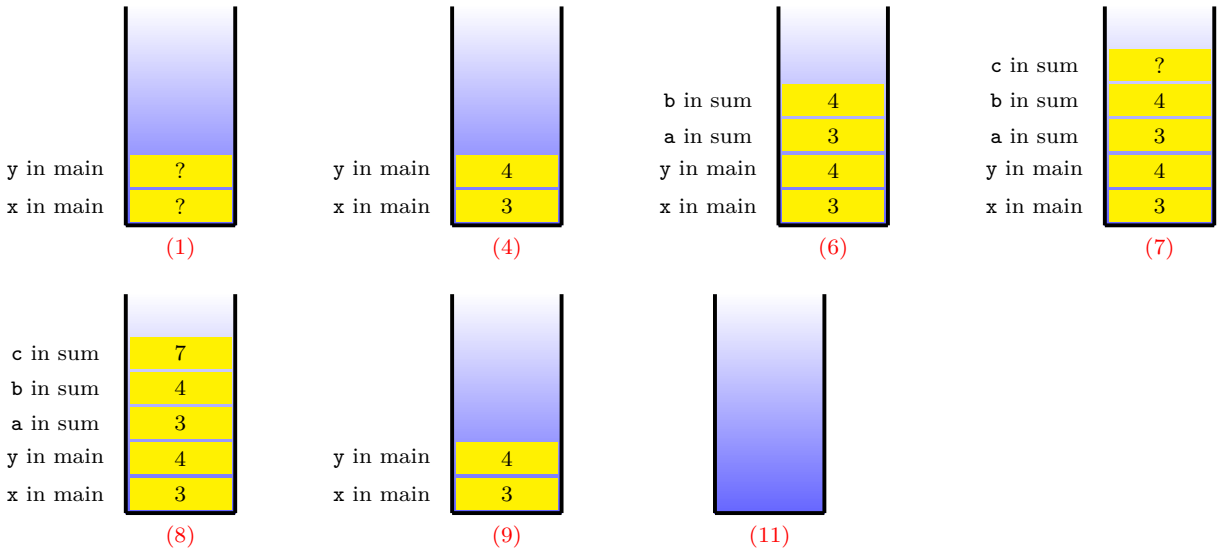
Example

Code Segment

```
#include <stdio.h>
int sum(int a, int b)(6)
{
   int c;(7)
   c=a+b;(8)
   return c;(9)
}
int main()
{
   int x, y;(1)
   printf("Enter 2 integers\n");(2)
   scanf("%d%d", &x, &y);(3)
   printf("The sum is %d\n", sum(x, y)(5));(10)
   return 0;(11)
}
```

Screen/Console

```
Enter 2 integers      (2)
3
4 }(4)
The sum is 7          (10)
```



In an assignment or an exam
when you are asked to trace the execution of a program
in the "programming model",
draw with the following format

It is fine to hand-draw the pictures (neatly),
take a photo, and submit the photo

Format

- The code (C source code) is placed in the Code Segment.
- All screen displays are placed in the Console.
- A "step" of program execution refers to
  - a line (or block) of the code which causes a change in the program memory or a change in the display on the Console,
  - an input (or input block) from the keyboard which causes a change in the program memory,
  - a calling of a non-system-built-in function, or
  - a calling of an stdio function (e.g., scanf)
- All declaration statements in the same function can be considered as one step.

Format (Continued)

- Each step should be sequentially labelled by number 1, 2, ..., following the order of the program execution.
- If a step results in a state change in a memory segment, the resulting state in the memory segment should be drawn and labeled with the step number.
- If the display of a block of text on the Console results from executing a step, the display should be labelled by the step number.