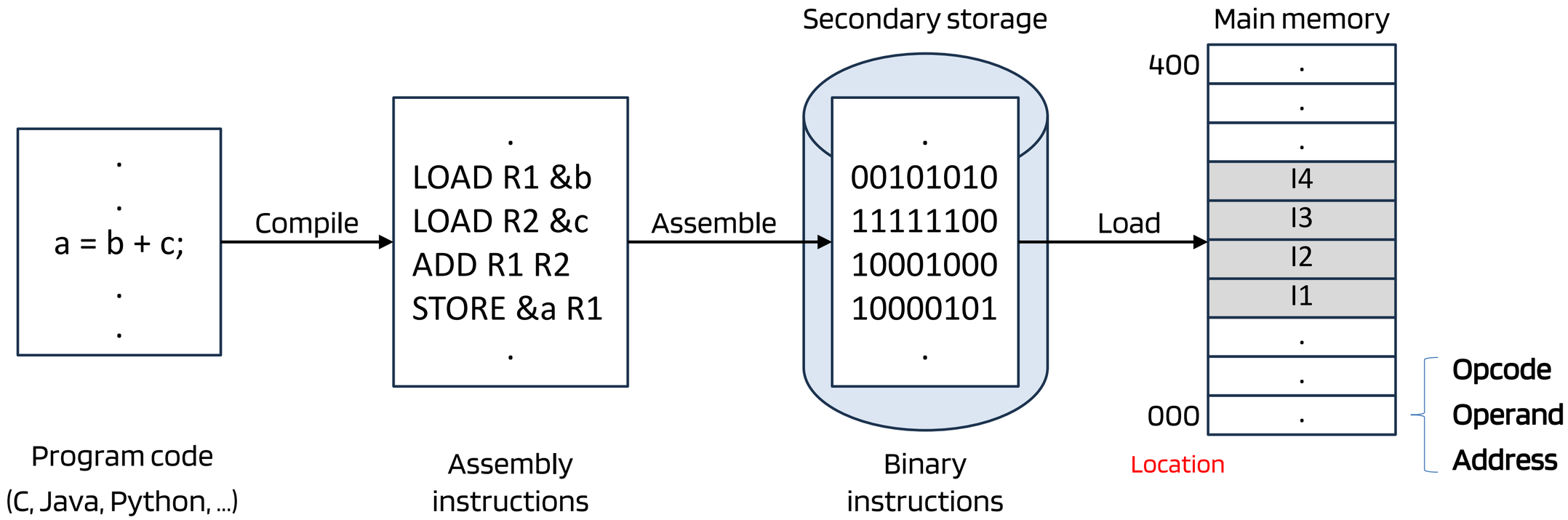
CPE3201

Process Management

Dr. Pongrapee Kaewsaiha

Computer Engineering
Suan Sunandha Rajabhat University

How a program is executed

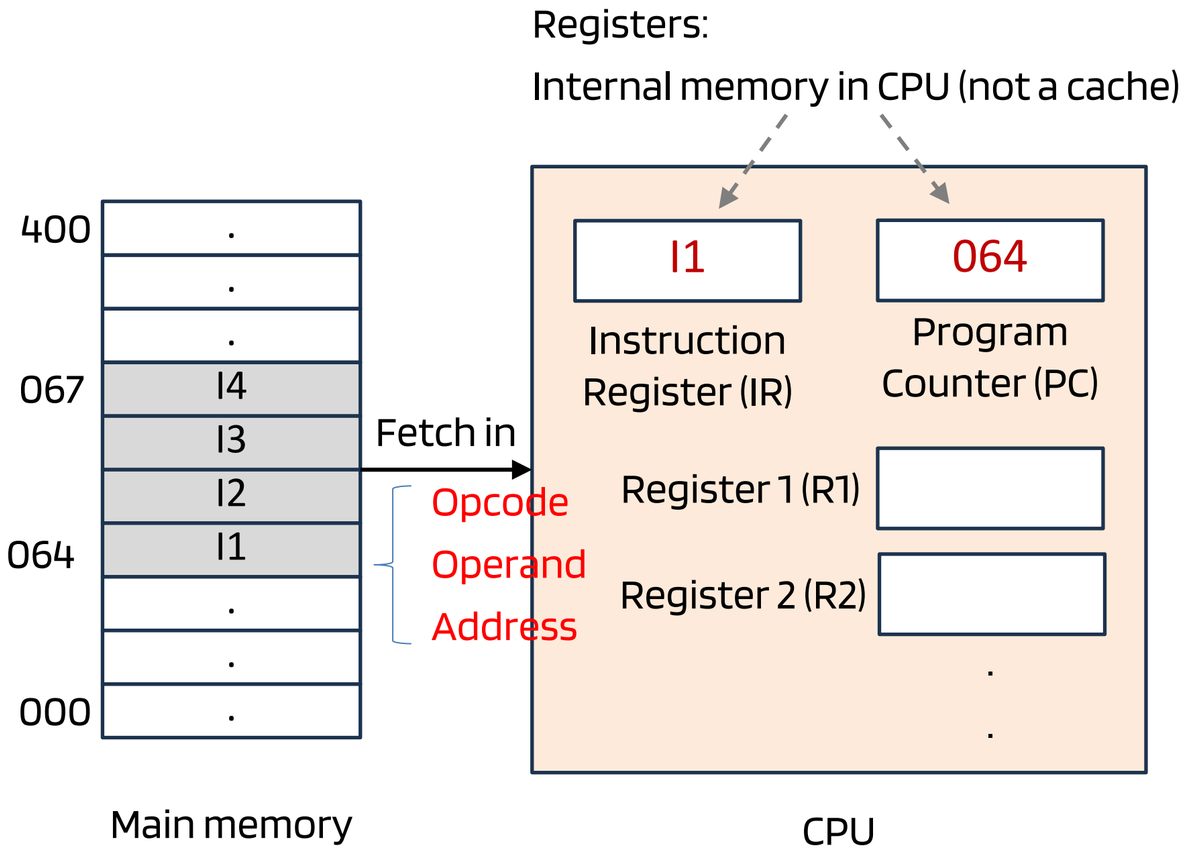


Opcode – The operation to be performed.

Operand – Variables on which an operation is performed.

Address – Memory location of instruction/data.

* This slide is only about "instruction," data will be mentioned later.



Control Unit (CU)

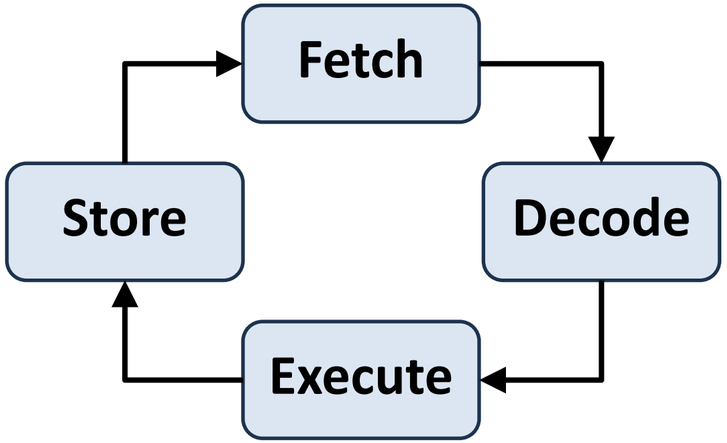
- Controls data sent through CPU's various components.
- Receives and transmits control signals from other devices.
- Interprets commands and controls CPU time.
- Decode, fletch, carry out the order, and store the results.
- Interprets directions and directives.

Arithmetic Logic Unit (ALU)

- Performs all the basic arithmetic operations (+, -, *, /, ...) and logical operations (AND, OR, ...).

- Program instructions and data must be first loaded in memory before CPU can access them.
- PC will always be incremented by 1 address location (e.g., 064 → 065 → 066 → ...).
- CPU cannot directly fetch instructions from secondary storage.

Instruction cycle



Fetch – Fetches instruction and data from RAM.

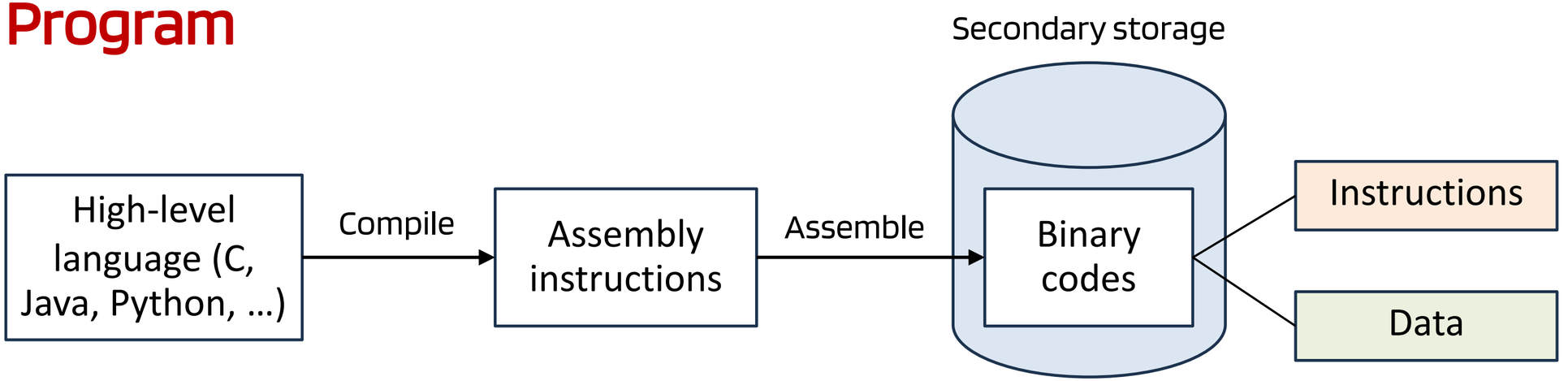
Decode – Decodes the instruction.

Execute – Executes instruction and operates on data.

Store – Stores output data in RAM.

Program vs Process

Program



.c .cpp .java .py

.exe .cgi

Static – Fixed size

Dynamic – Variable size

Static and dynamic data

```
int a;
main() {
  int b, c;
  b=1;
  c=2;
  a=b+c;
---
  int *p=(int*)malloc(4);
---
  Class A obj=newClassA();
---
}
```

Global variable (Static)

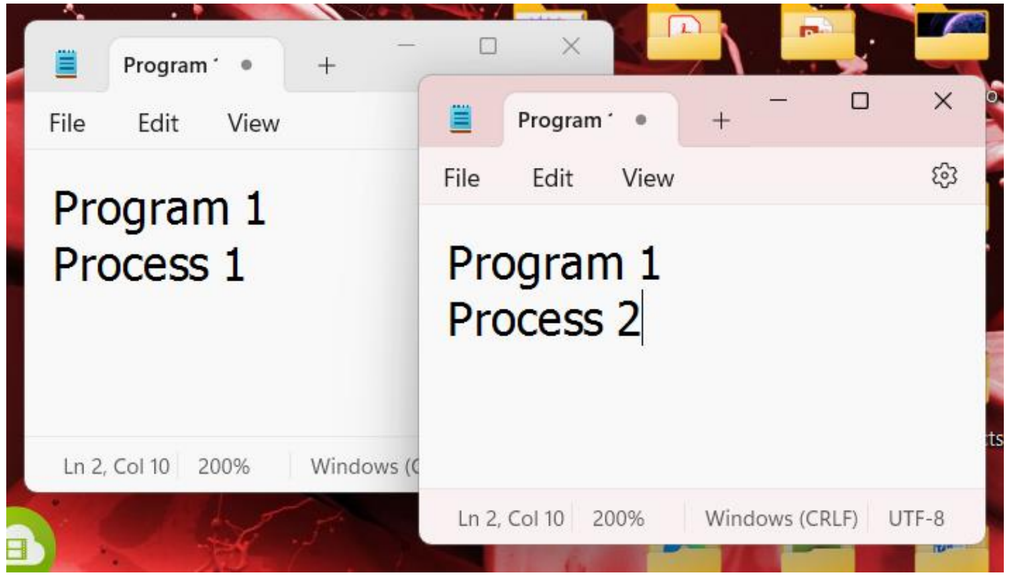
Local variables (Static)

Dynamic

Dynamic

Program & Process

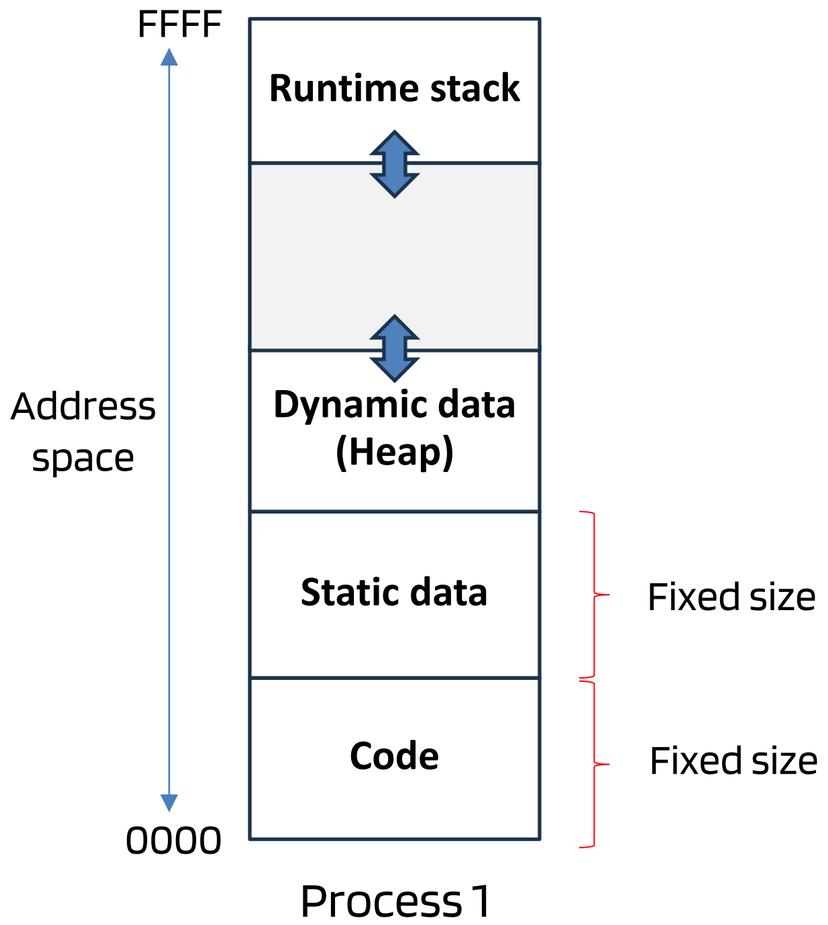
- **Program** is stored in a secondary storage. It does not use CPU.
  - Example: notepad.exe in C:\Users\Program Files
- **Process** is a program under execution, which is loaded in memory and utilizing resources.
- **Process** is an instance of program in memory.



Process structure

Address space

Memory allocated for a process

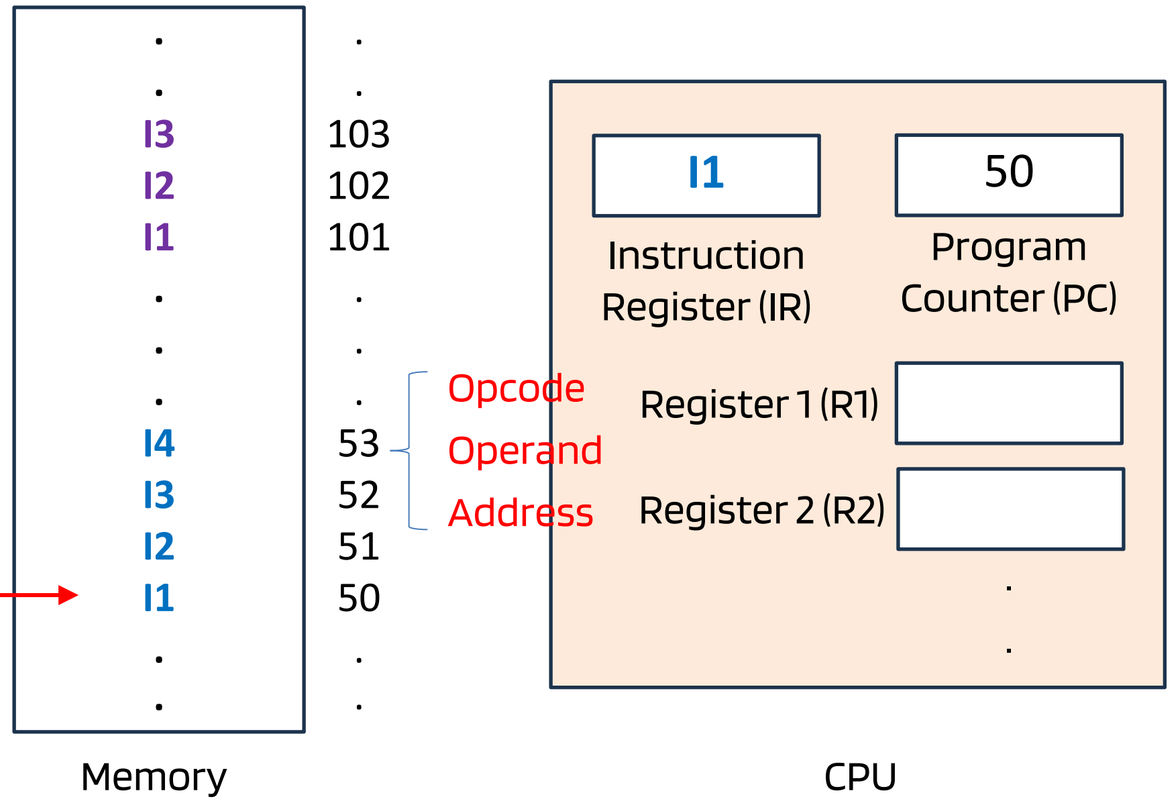


Code section

- Binary instruction from programs are loaded in code section function-wise, contiguously.
- Each instruction has an address, opcode, and operands
- Program Counter (PC) is initialized with the first instruction's address of main() function.

```
main() {
  int a=1;
  int b=2;
  int c=a+b;
  funcX();
}

funcX() {
  int d=20;
  int e=10;
  int f=d-e;
}
```



Once the code is loaded, the program must tell the CPU the address of the first instruction in the main function.

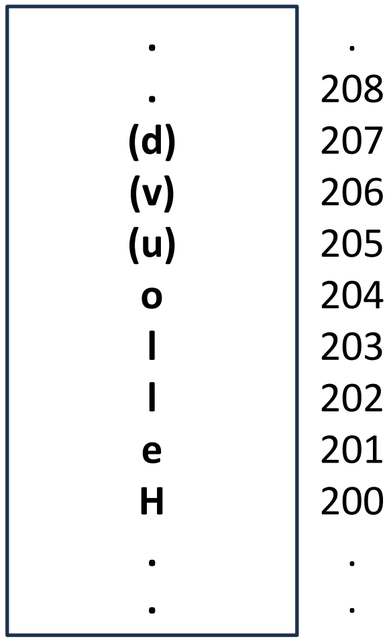
Address does not show the actual size.

Static data section

- Variables declaration.
- Declaration + Assignment.

Global variables

```
#define t "Hello"
int u;
main() {
  int v;
}
funcY() {
  static float d;
}
```



Memory

Variables in C

Char: 1 byte = 8 bit

(e.g., H = 1001000)

Int: 2-4 byte

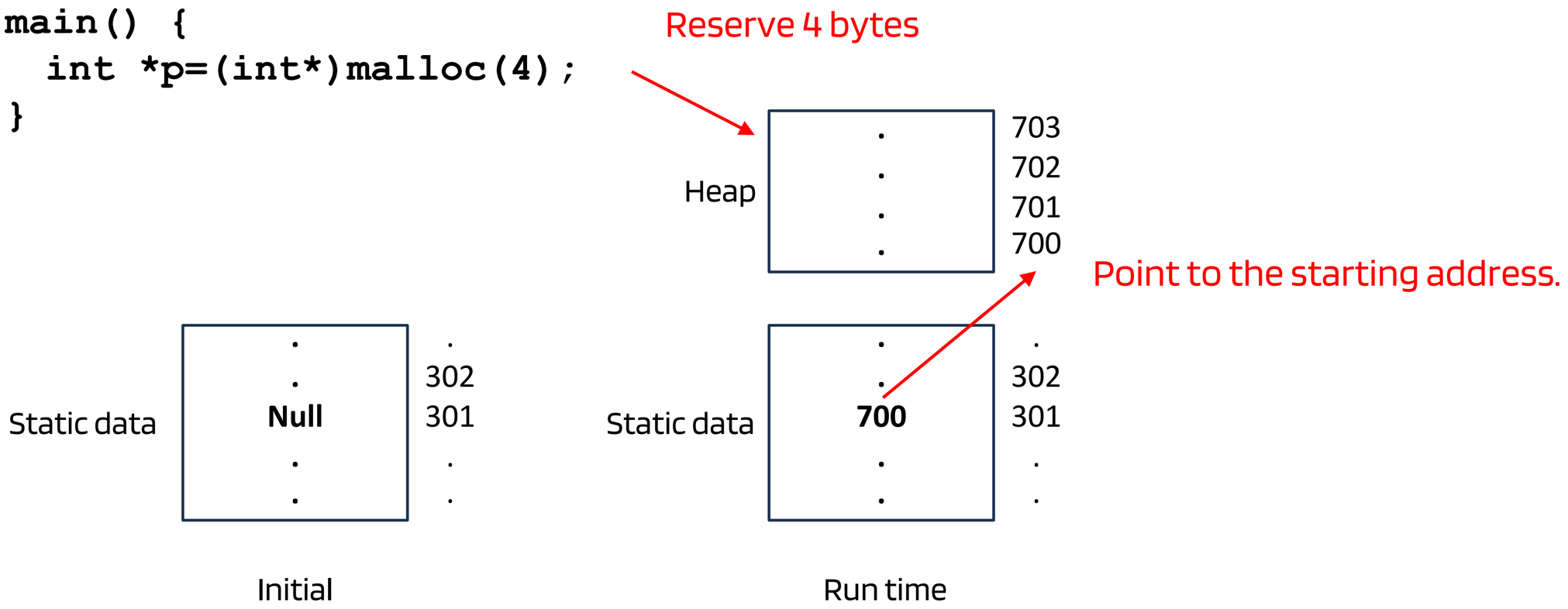
Float: 4 byte

Double: 8 byte

Dynamic (Heap) data section

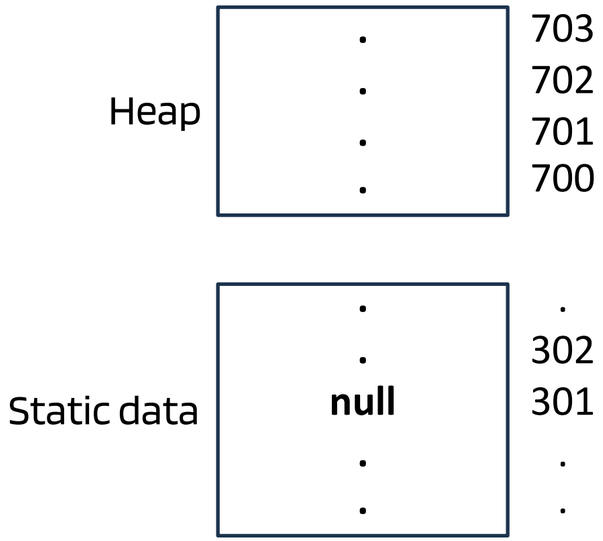
- Starts empty. Memory is allocated at run time.
- Programmer can decide the size and when to allocate/deallocate memory (e.g., malloc() calloc() and free in C).

```
main() {
  int *p=(int*)malloc(4);
}
```



Memory deallocation

```
main() {
  int *p=(int*)malloc(4);
  free(p);
  p=null;
}
```



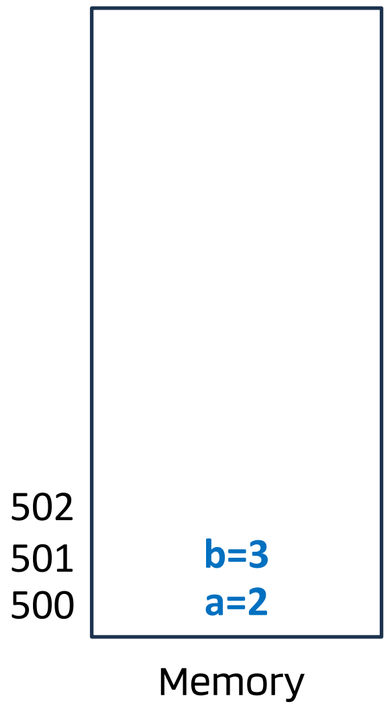
These addresses can be used again

Runtime stack

- It is a stack of Activation Record (AR) used to store local variables of function, return address, and parameter passed to the function.
- Before the program runs, the runtime stack is empty.

```
main() {
  int a=2;
  int b=3;
  funcX(a,b,10);
}
```

```
funcX(int p, int q,
int r) {
  int c;
  int d;
  return;
}
```

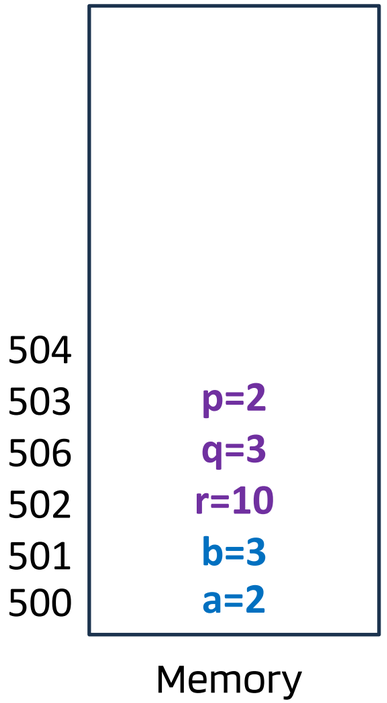


The memory is allocated in the same sequence as in the program.

AR of main()

```
main() {
  int a=2;
  int b=3;
  funcX(a,b,10);
}
```

```
funcX(int p, int q,
int r) {
  int c;
  int d;
  return;
}
```



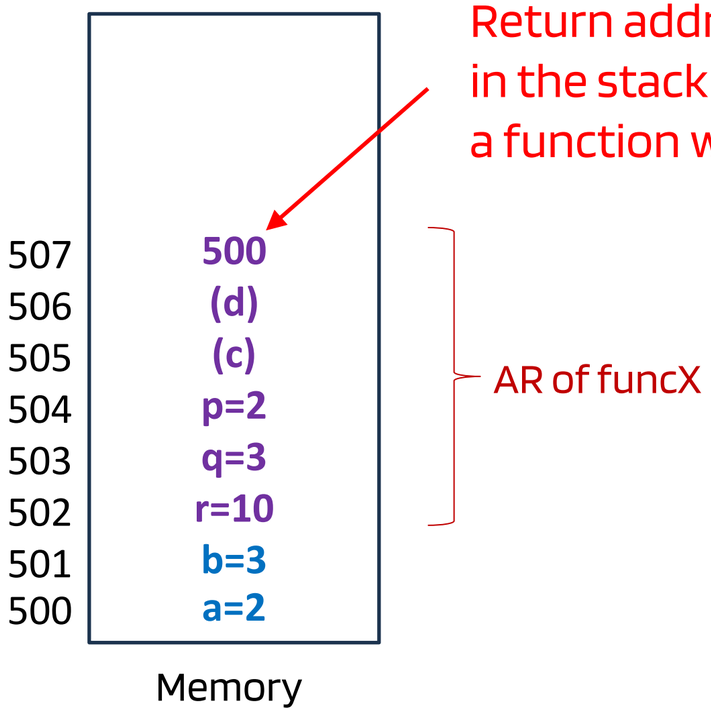
Main() is still active.

funcX is allocated on top of that.

Parameters are pushed into the stack from right to left in C (decided by a compiler).

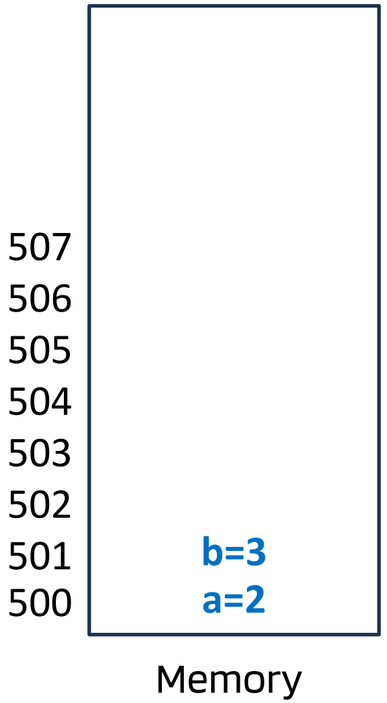
```
main() {
  int a=2;
  int b=3;
  funcX(a,b,10);
}
```

```
funcX(int p, int q,
int r) {
  int c;
  int d;
  return;
}
```



```
main() {
  int a=2;
  int b=3;
  funcX(a,b,10);
}
```

```
funcX(int p, int q,
int r) {
  int c;
  int d;
  return;
}
```



funcX returns value to the memory location.

Stack memory allocated to funcX is cleared.