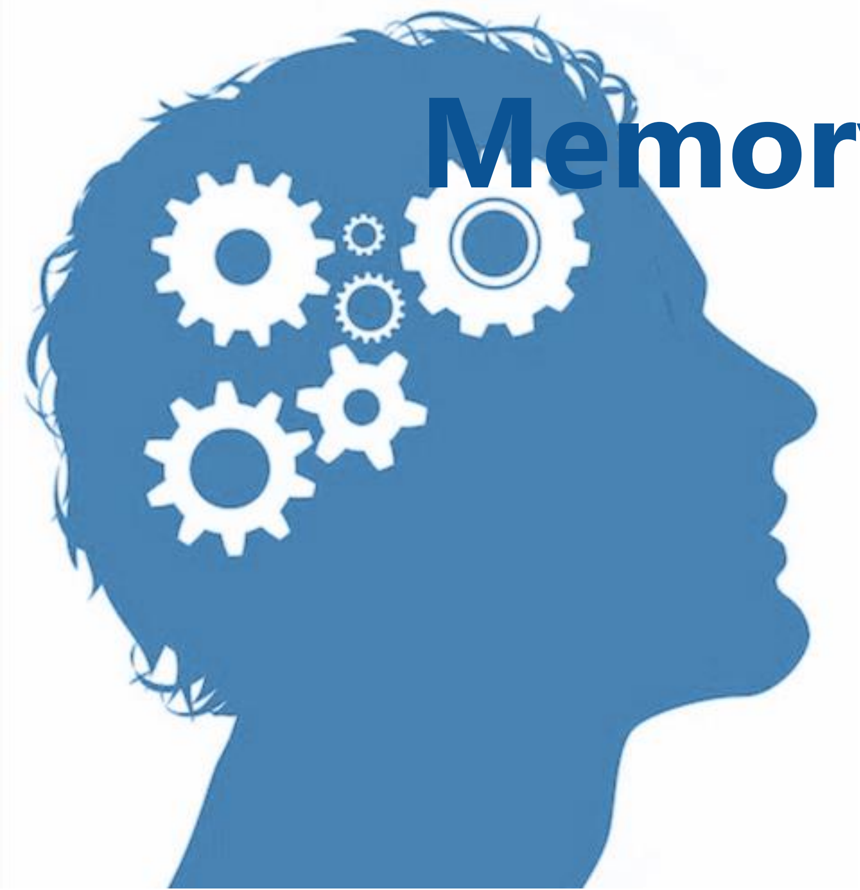


Memory management

CE-2101 Algorithms and Data Structures

Disclaimer / Descargo de Responsabilidad

Esta presentación corresponde a una guía usada por el profesor durante las clases. La misma ha sido modificada para ser utilizado en el modelo de cursos asistidos por tecnología. No es una versión final, por lo que la misma podría requerir todavía hacer algunos ajustes. Para aspectos de evaluación esta presentación es solo una guía, por lo que el estudiante debe profundizar con el material de lectura asignado y lo discutido en clases para aspectos de evaluación.

This presentation corresponds to a guide material used by the professor during classes. It has been modified to be used in the model of technology-assisted courses. It is not a final version, so it may still require some adjustments. For evaluation aspects, this presentation is only a guide, so the student should delve with the assigned reading material and what has been discussed in class.

Before we begin...

➔ What is the operating system?

➔ What are the **main functions** of the operating system?

Before we begin...

- ➔ An operating system is **software**. Is a software layer that sits on top of the hardware.

- ➔ Main functions (top level):
  - ◆ Provide an API for developers that abstracts resources
  - ◆ Manage all the resources of the machine

Before we begin…

➔ Main functions:

- ◆ Process management
- ◆ Memory management
- ◆ File & Disk management
- ◆ I/O management

Before we begin...

- ➔ You'll learn more about operating systems and memory management in CE-4303
- ➔ For now, it is good to have a clear context of the topic before we begin
- ➔ We will focus on memory management from the **point of view of coding**, but with an overview on how the OS manage it

Memory Management

- ➔ What is memory management?
- ➔ What involves memory management?
- ➔ Take a guess, how does the OS manage the memory?

Memory Management

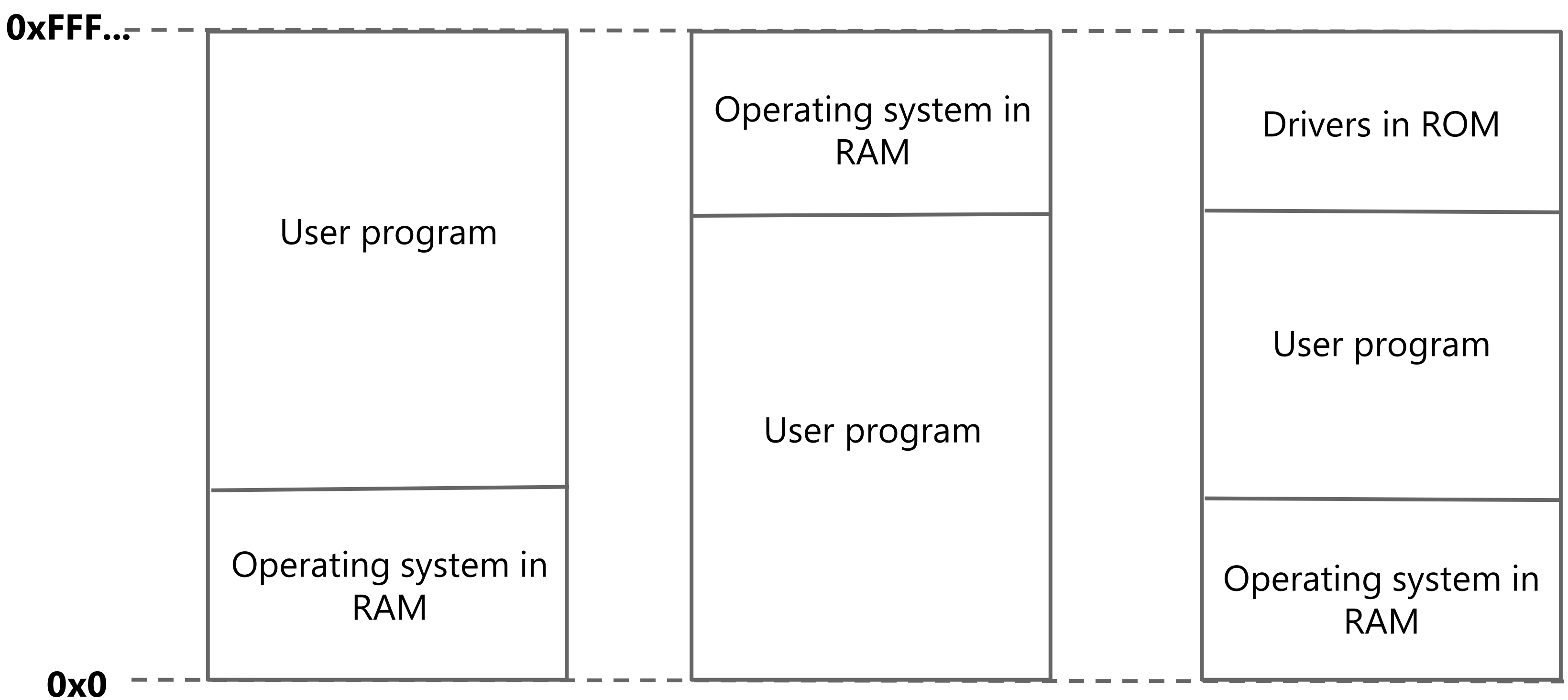
- ➔ There have been **different schemes or approaches** on how to manage memory properly
  - These schemes have changed or evolve to meet more strict requirements, specially related to **multiprogramming**
  - Ranging from not memory management at all - just one bucket of memory - to the current concept of virtual memory

Memory Management

- ➔ RAM memory is a valuable resource and it has to be handled carefully
- ➔ Even though it is **cheaper and more available** than before
- ➔ The part of the operating system that manage the memory is called **memory manager**.

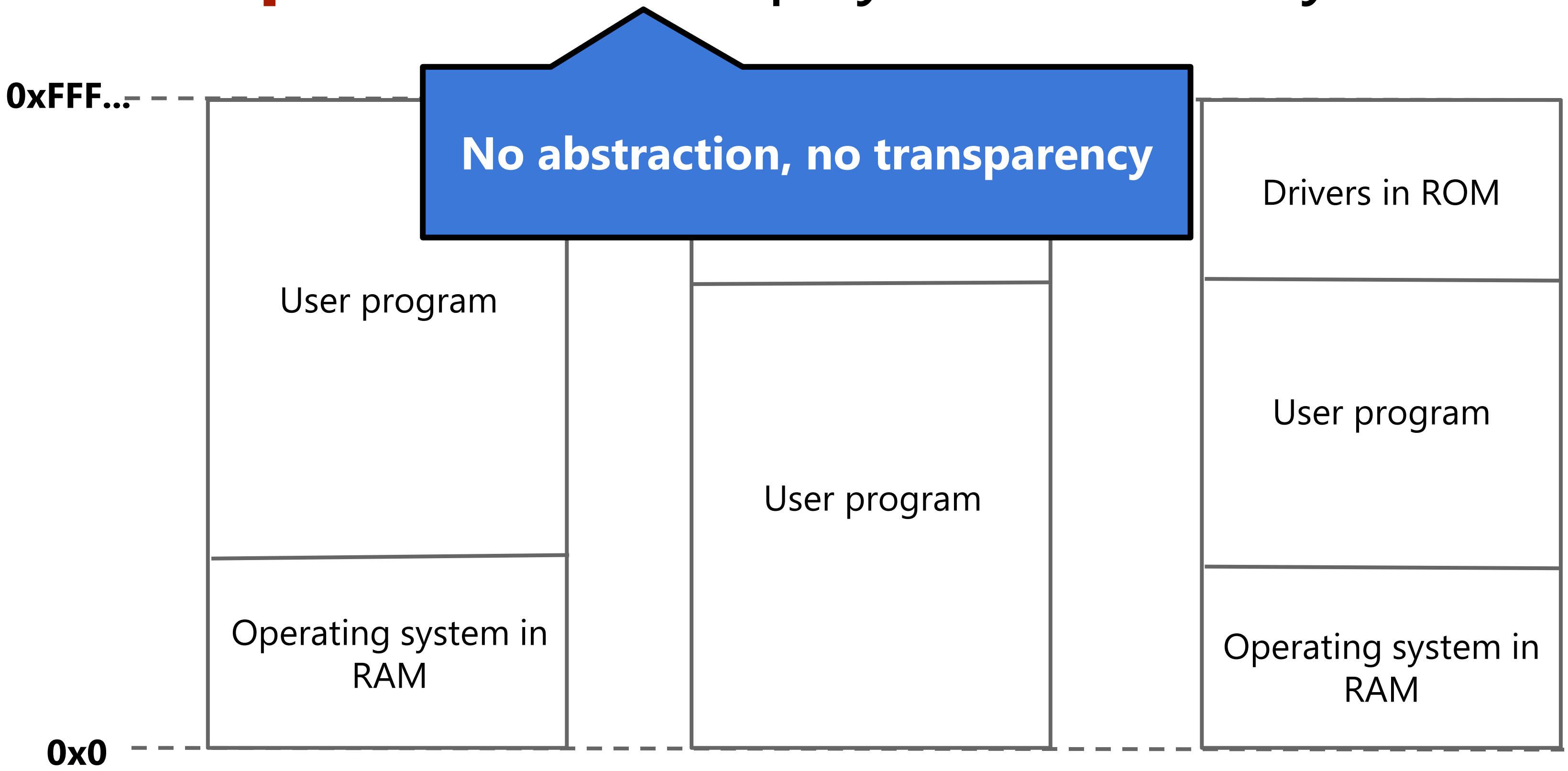
Memory Management Schemes

➔ **No separation** from physical memory



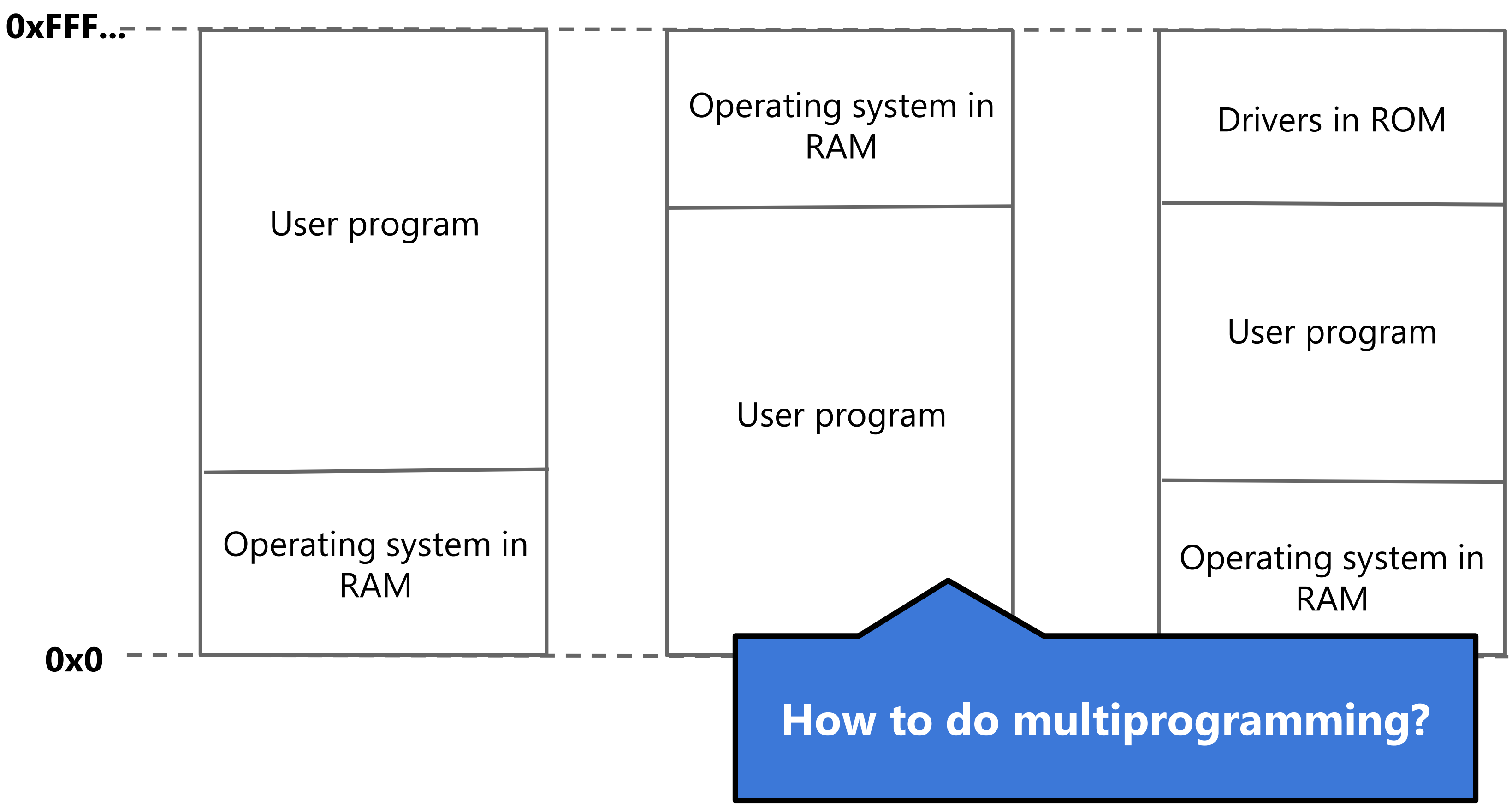
Memory Management Schemes

➔ **No separation** from physical memory



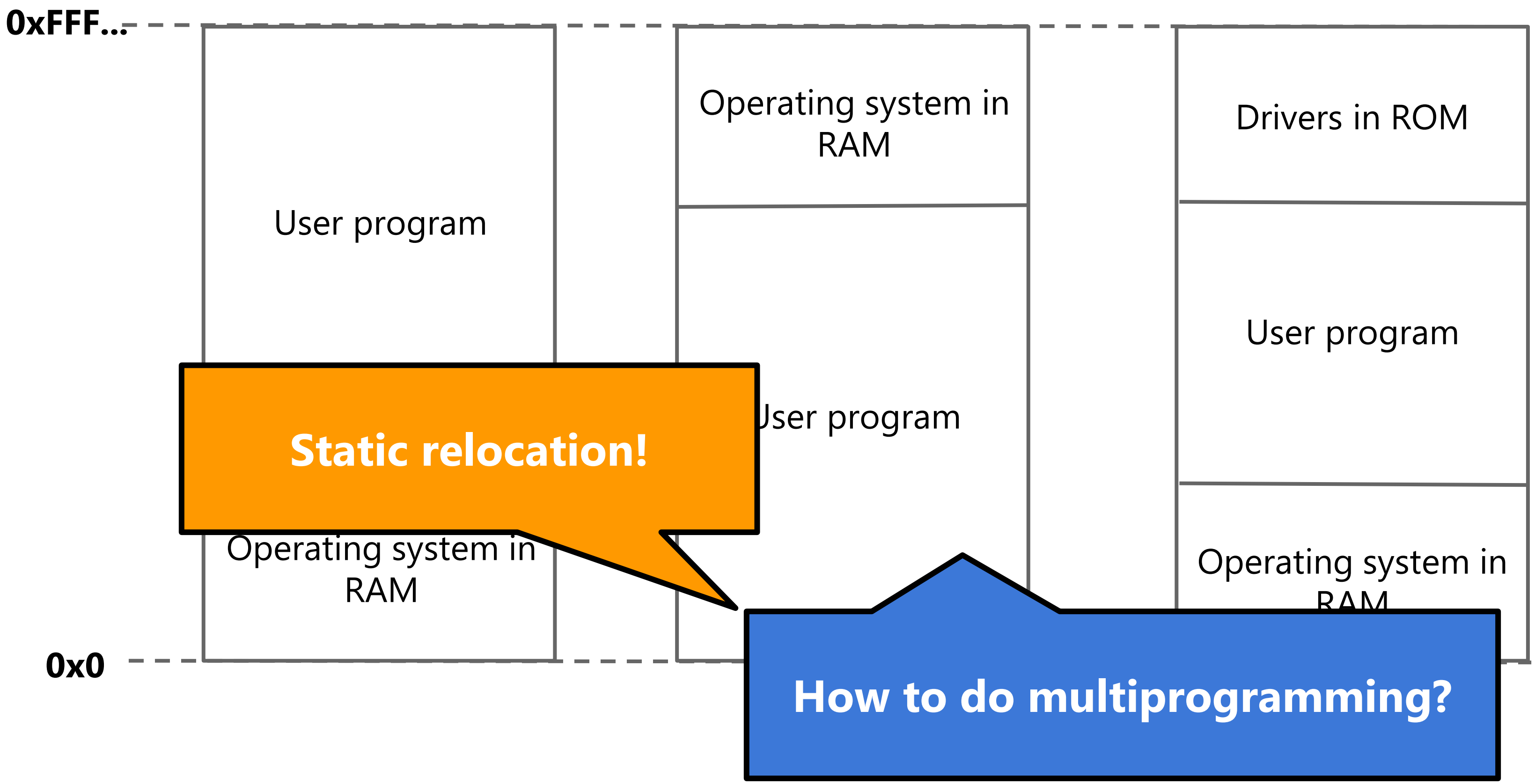
Memory Management Schemes

→ **No separation** from physical memory



Memory Management Schemes

➔ **No separation** from physical memory



Memory Management Schemes

- ➔ In time there were new schemes to manage memory:
  - ◆ **Address spaces**: Each program gets a section of the memory assigned to itself and no one else
    - Base and limit registers
    - Dynamic relocation
    - Swapping for multiprogramming

Memory Management Schemes

- ➔ In time there were new schemes to manage memory:
  - ◆ **Address spaces**: Each program gest a section of the memory assigned to itself and no one else
    - Base and limit registers
    - Dynamic relocation
    - Swapping for multiprogramming

In this approach the complete program had to be on memory

The size of the programs is increasing

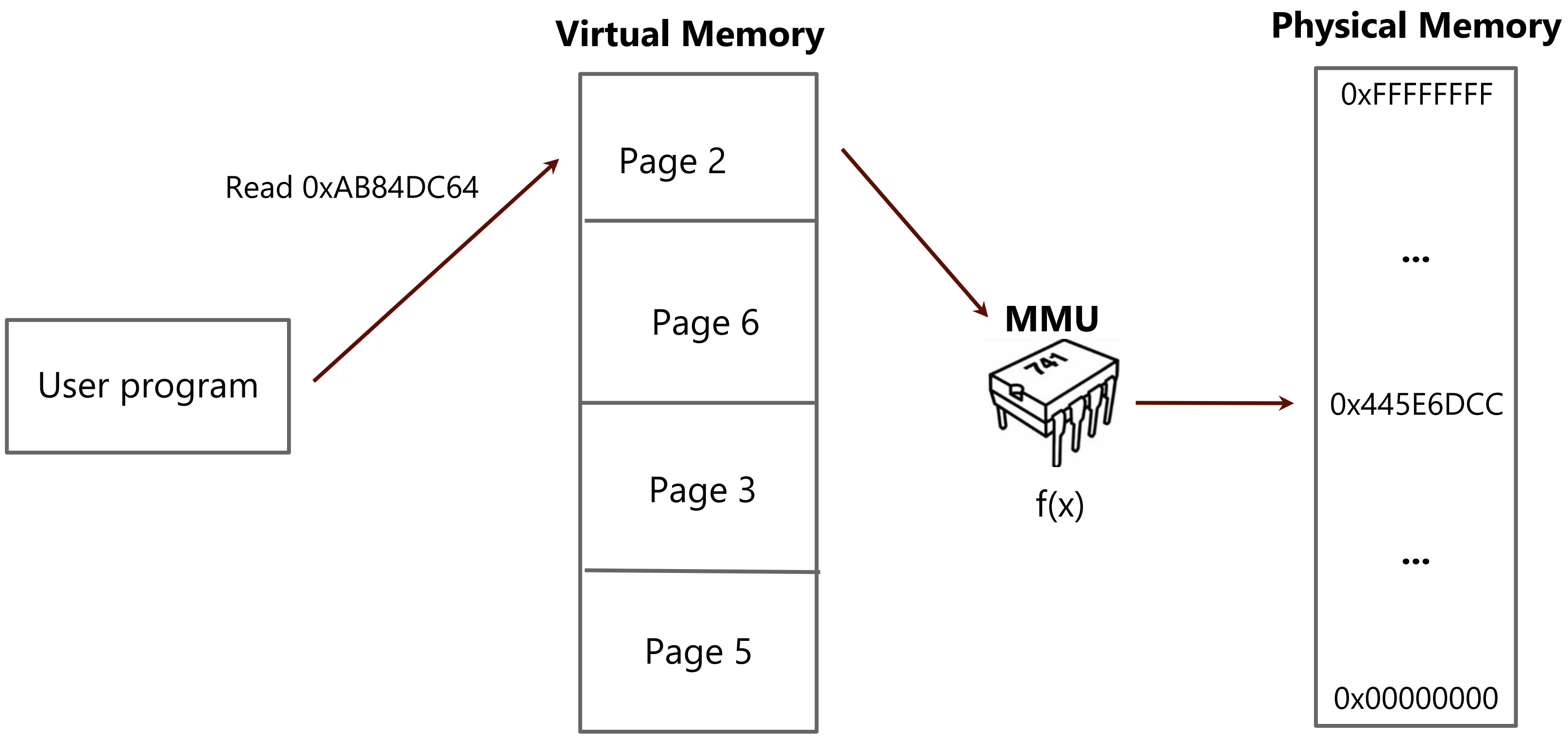
Memory Management Schemes

➔ How can a program of 2 GB run in a computer with only 512 MB of RAM?

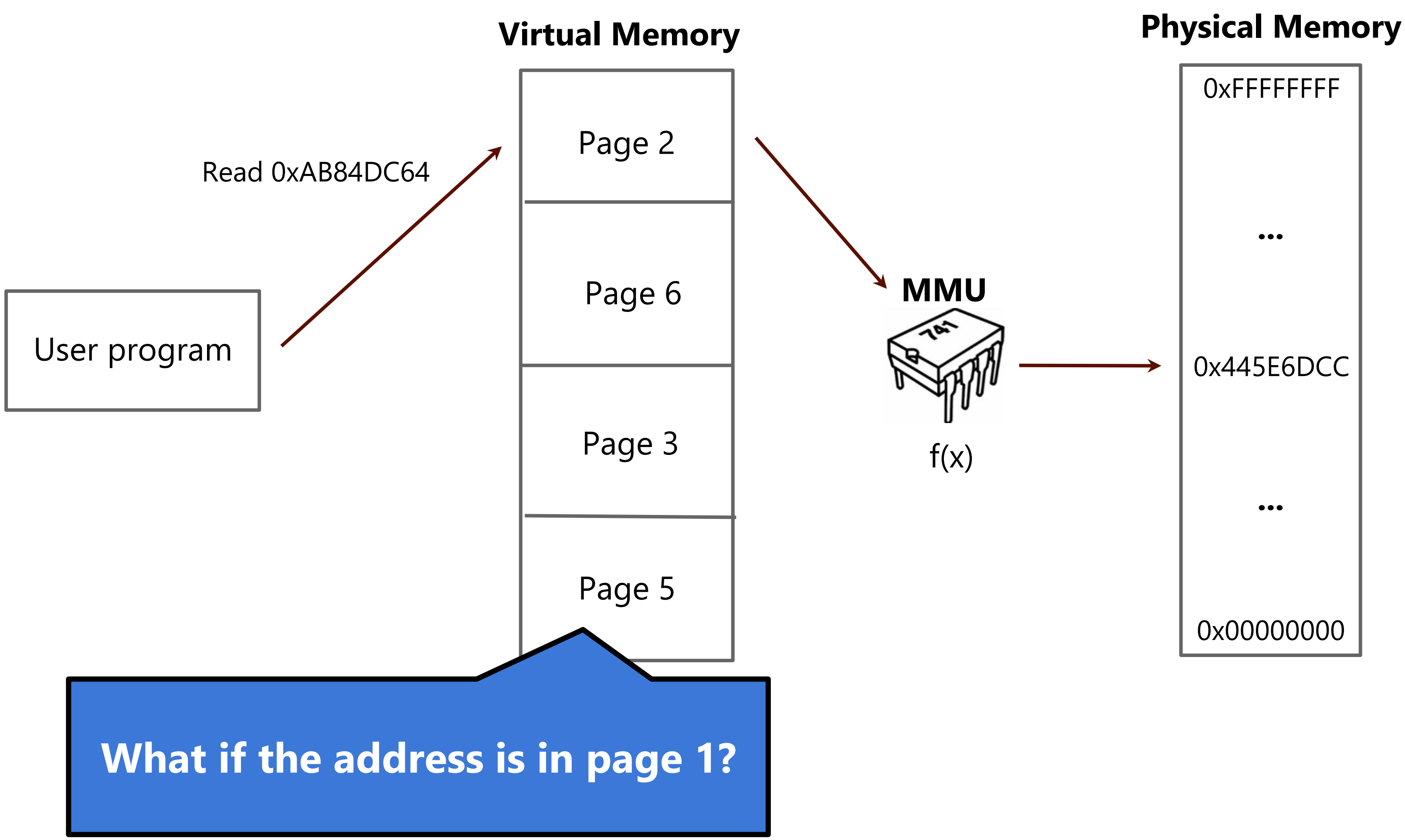
Memory Management Schemes

- ➔ **Virtual memory**: Each program has its memory space which is separated in pages of fixed size.
- ➔ Each page is a small range of memory addresses.
- ➔ The program does memory operations **transparently**

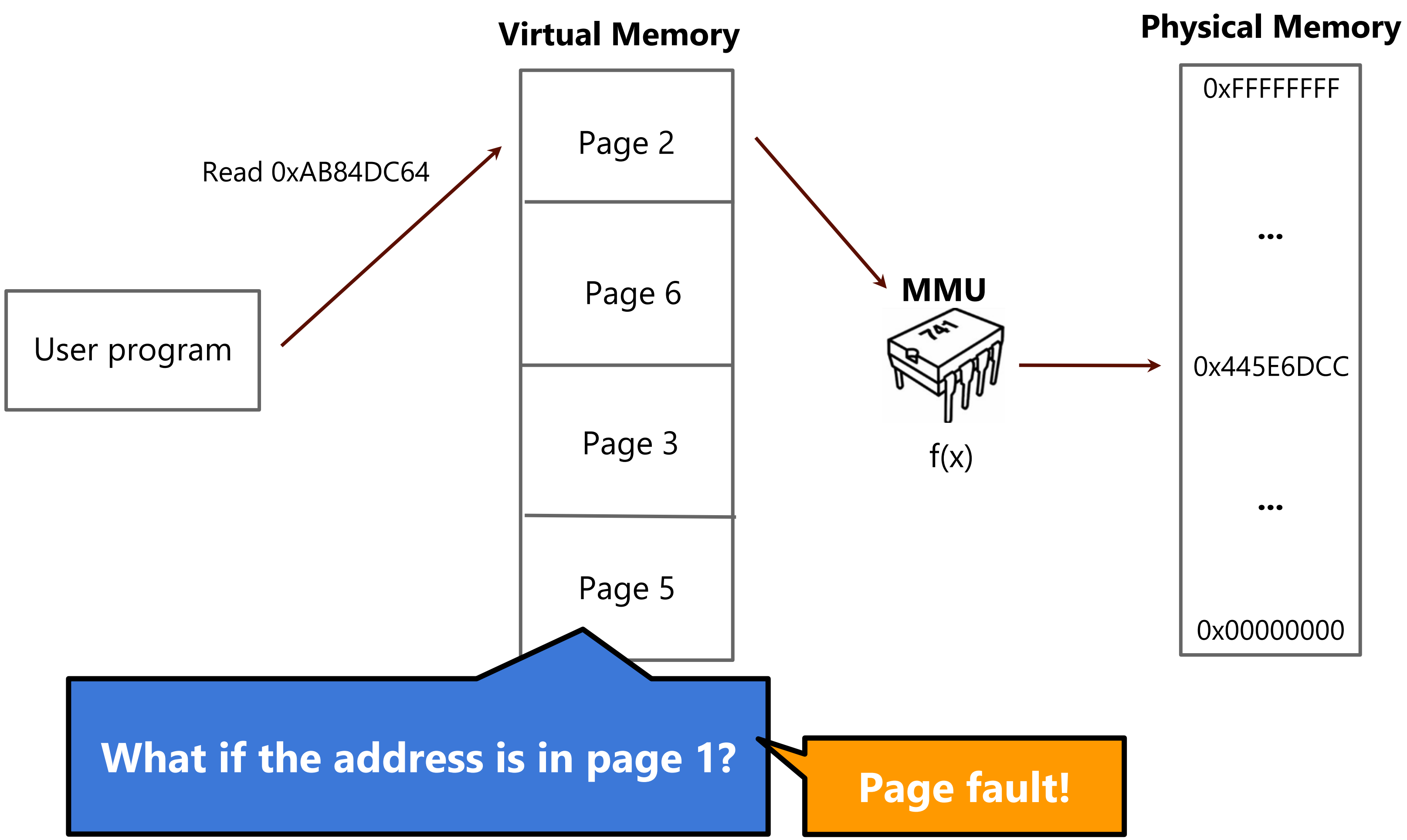
Memory Management Schemes



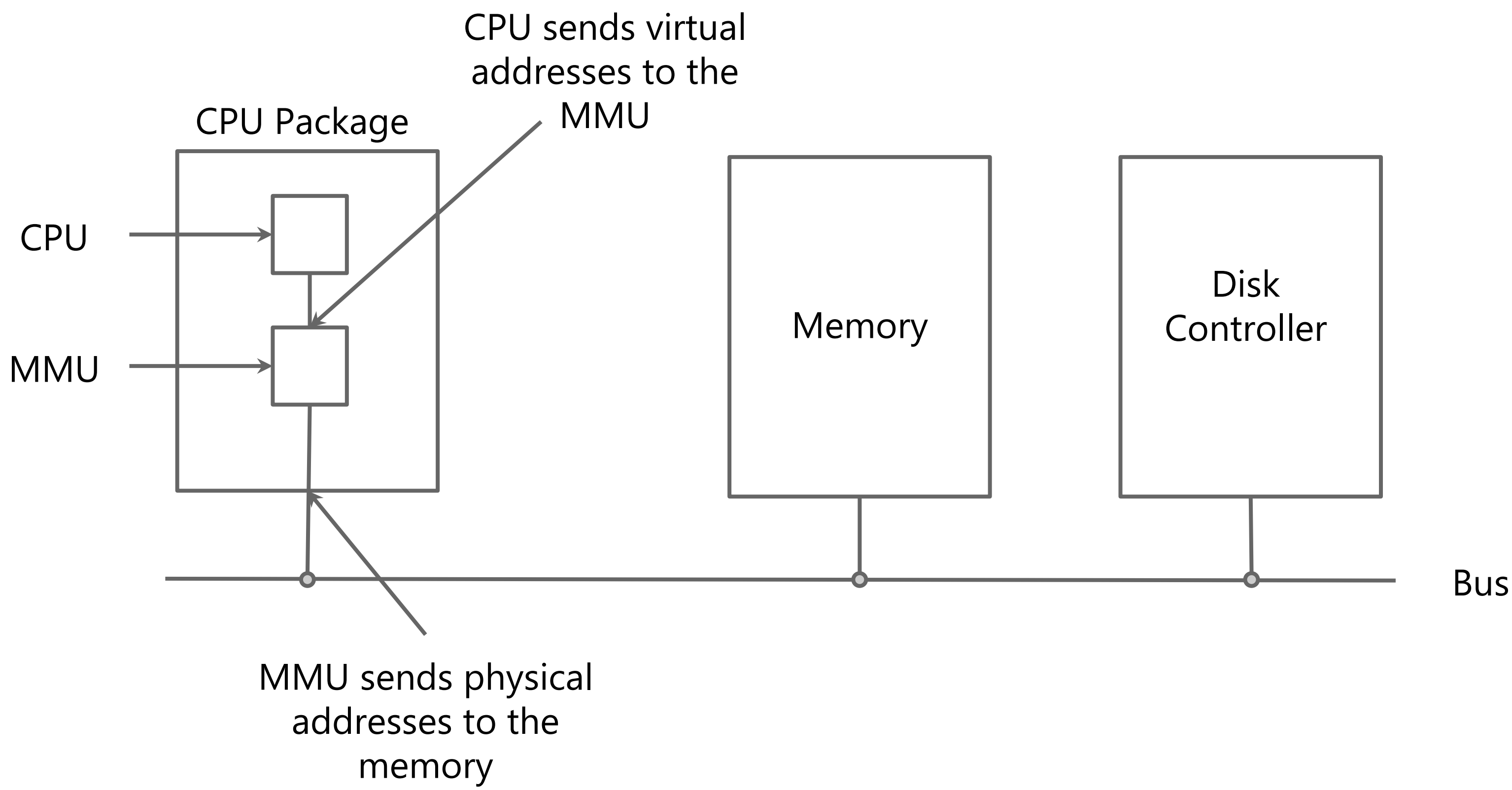
Memory Management Schemes



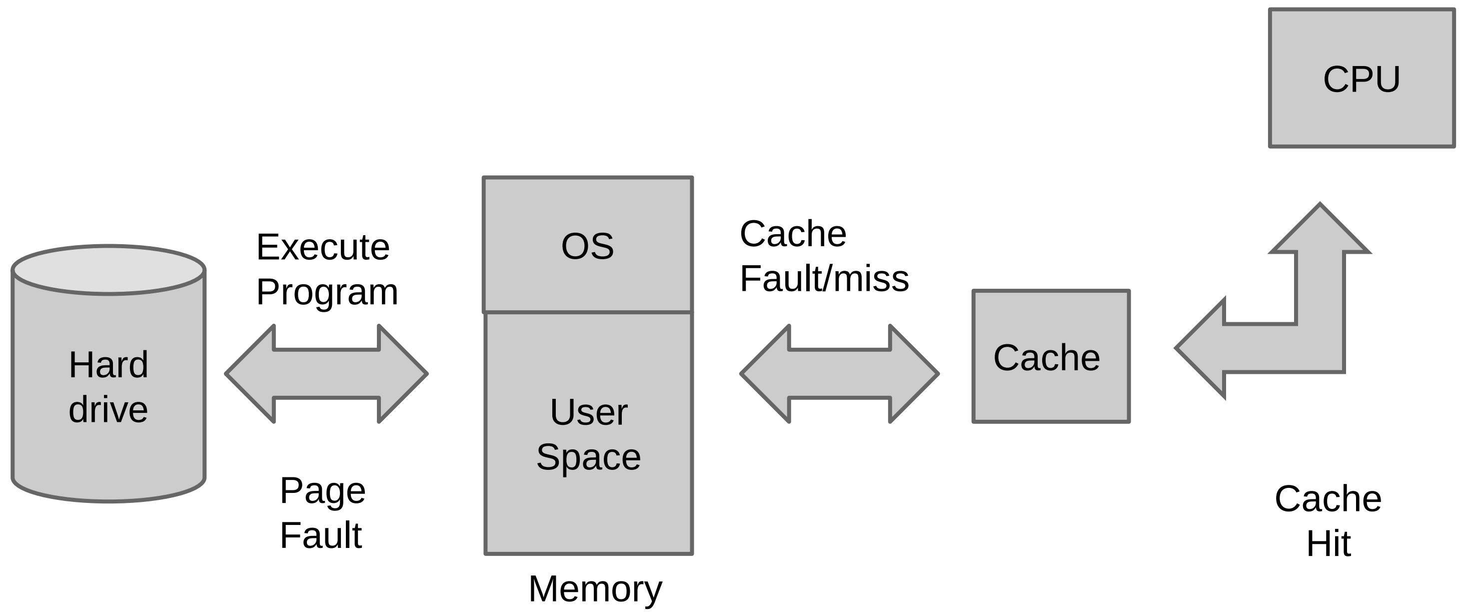
Memory Management Schemes



Memory Management Schemes



Memory Management



Memory management: program level

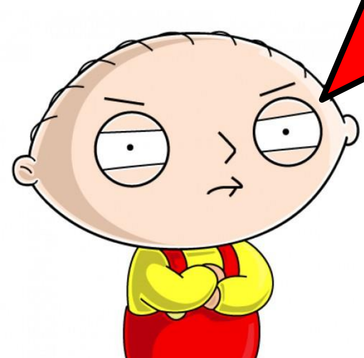
- ➔ We've seen a quick overview of how memory is managed at OS level.
- ➔ Let's focus on memory management from the program's point of view

Memory management: program level

- ➔ We've seen a quick overview of how memory is managed at OS level.

- ➔ Let's focus on memory management from the program's point of view

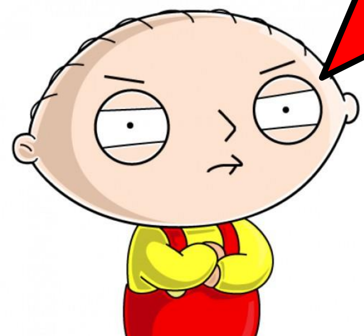
Why are we learning about memory in a data structures course?!



Memory management: program level

- ➔ We've seen a quick overview of how memory is managed at OS level.
- ➔ Let's focus on memory management from the program's point of view

Why are we learning about memory in a data structures course?!



Even the most elegant and efficient data structure can be disempowered if memory is handled wrong

Memory management: program level

→ How is the memory of a program structured?

→ What is the *heap*?

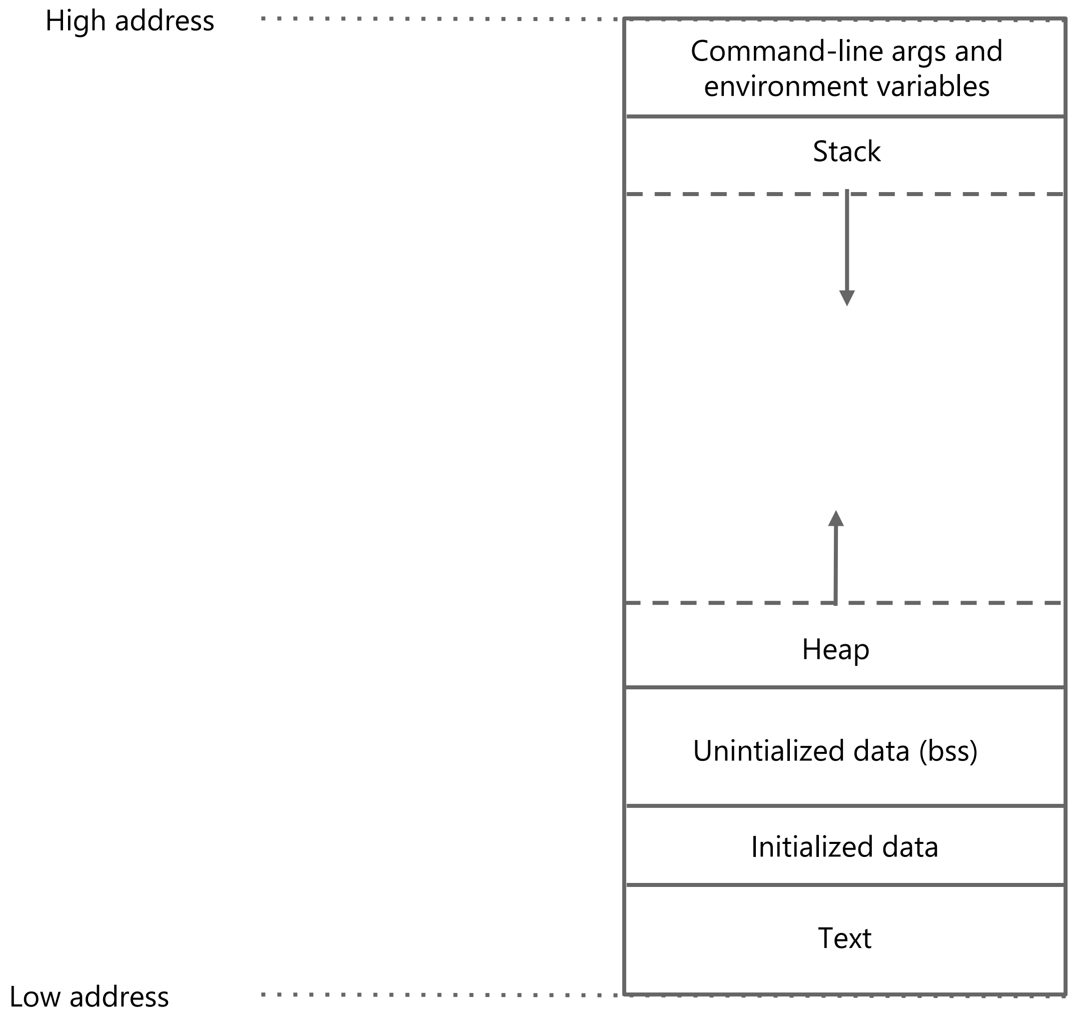
→ What is the *stack*?

→ Pointers? References?

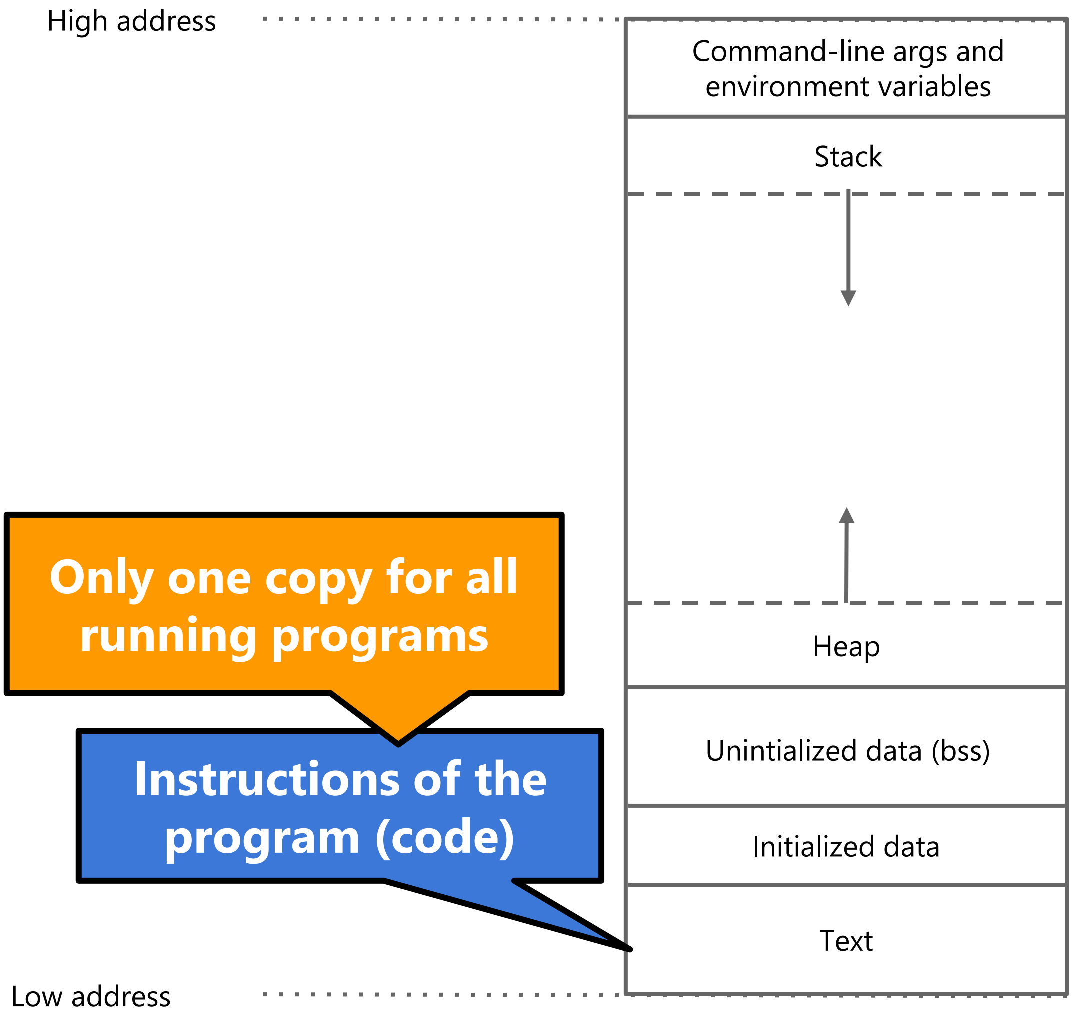
Memory management: program level

- ➔ A program in execution (process) has a well known structure for its memory
- ➔ This **memory layout** may vary depending on the programming language, but it is similar conceptually
- ➔ Let's see how look the memory layout of a C program when its running

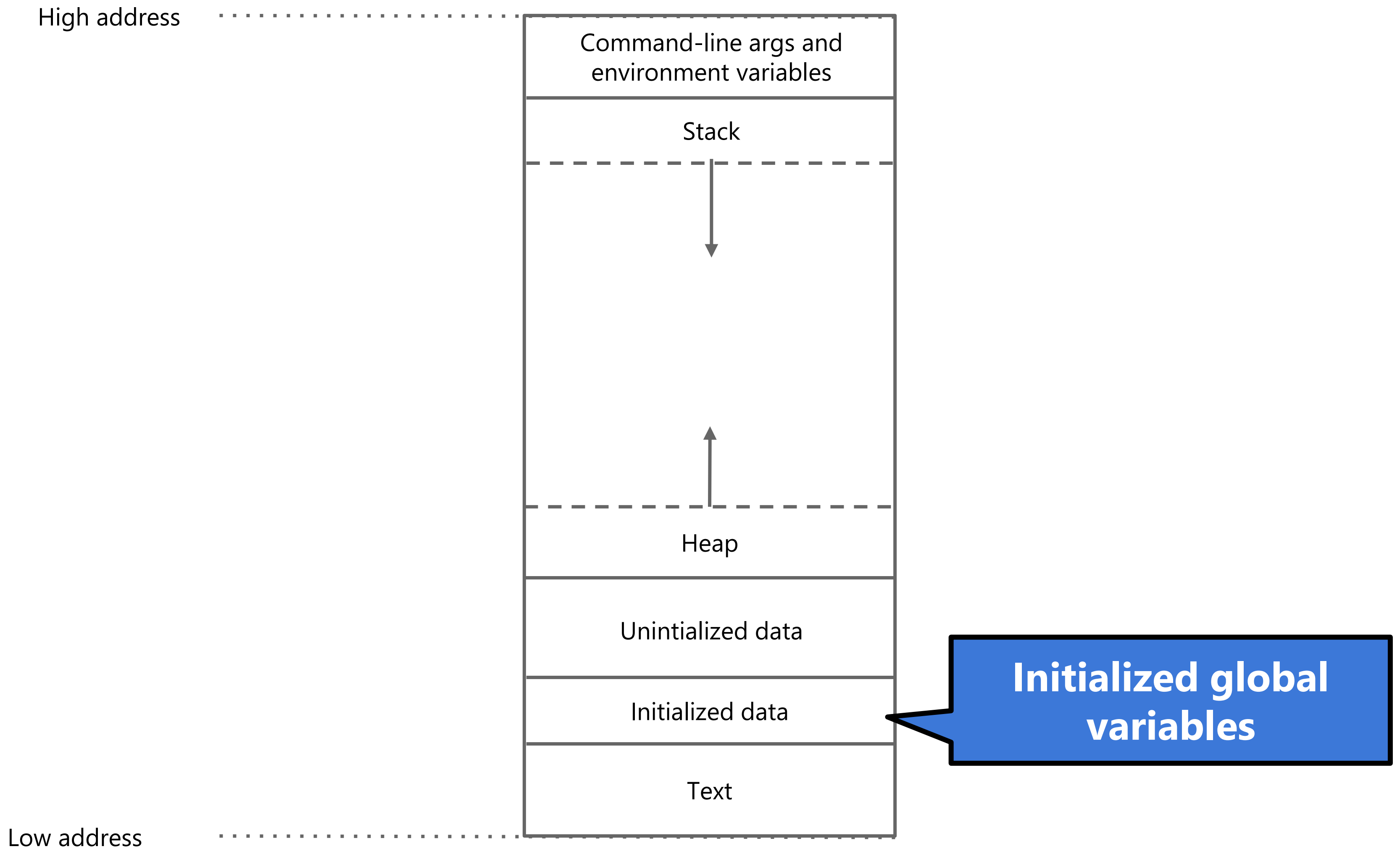
Memory management: program level



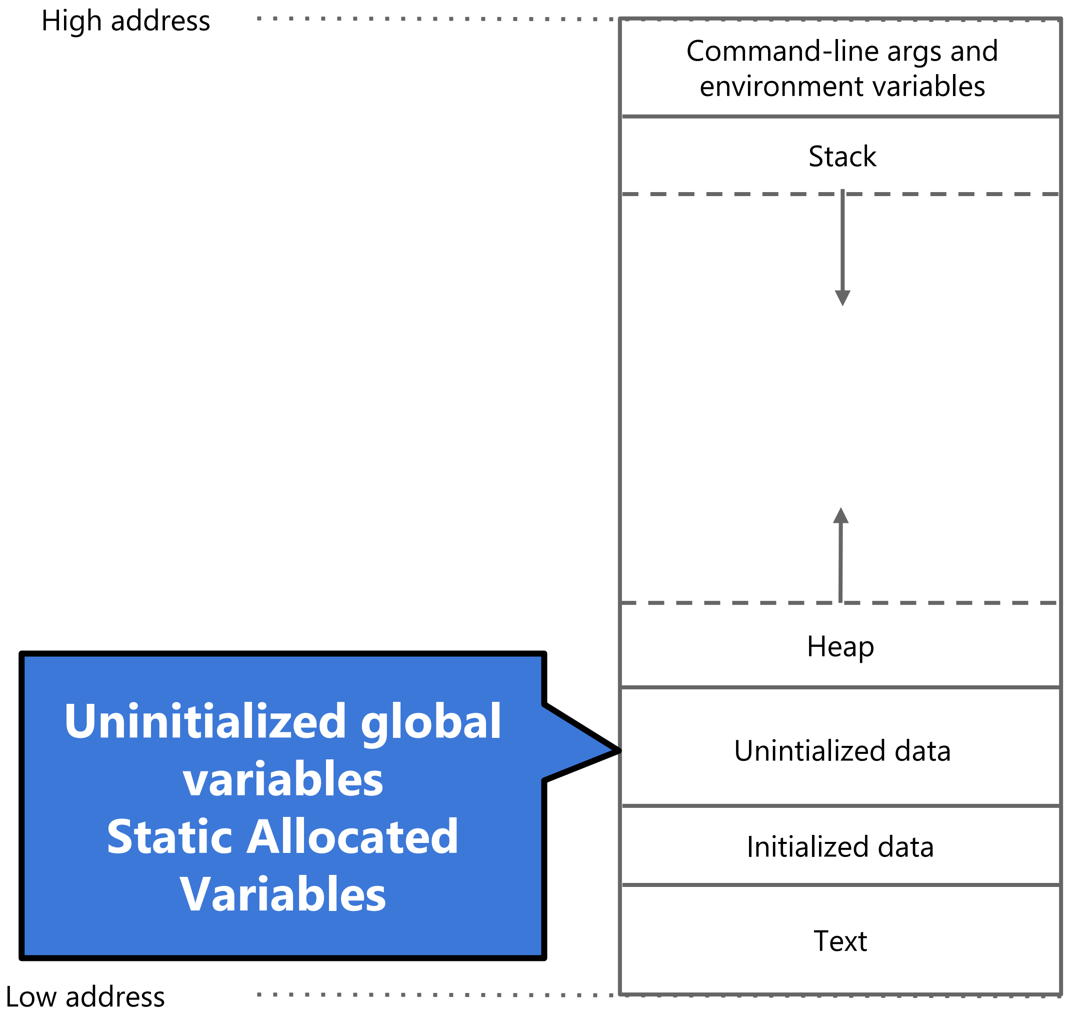
Memory management: program level



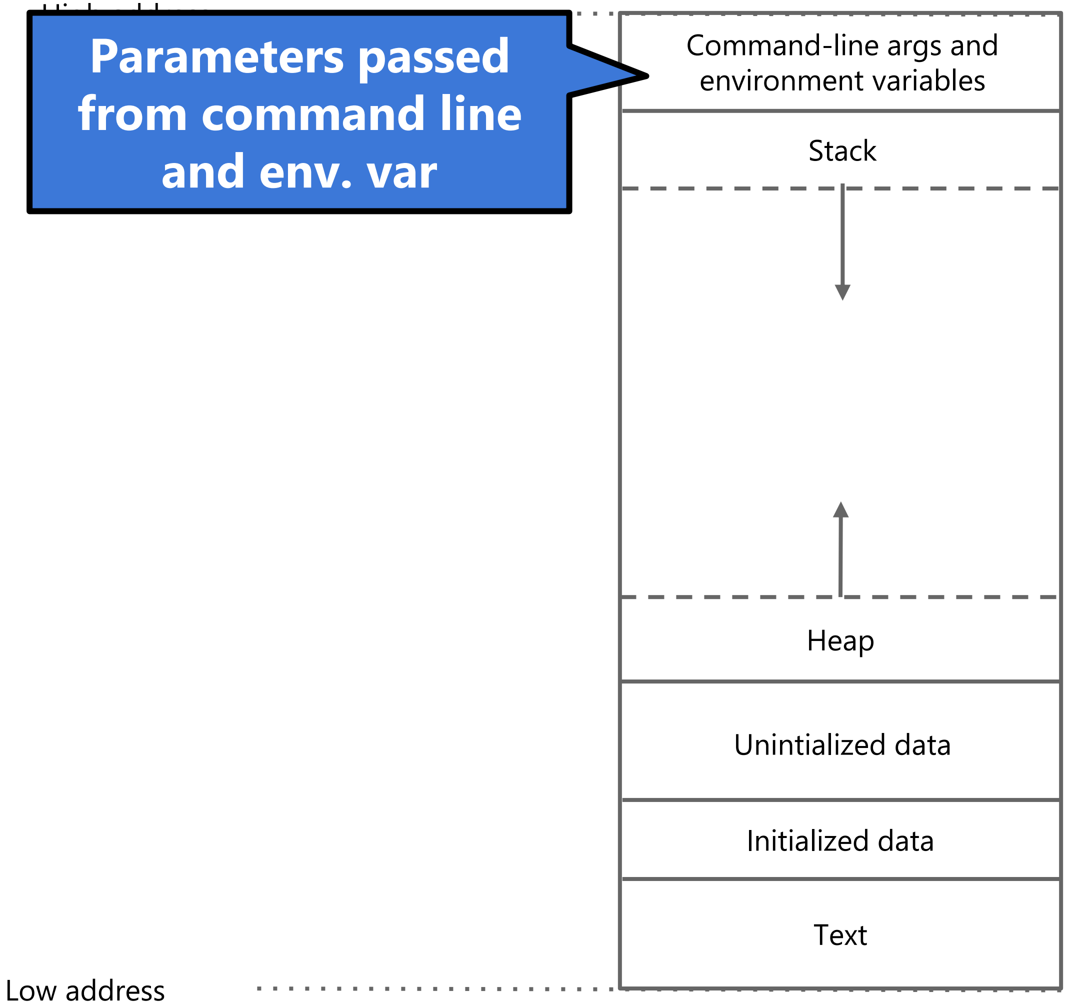
Memory management: program level



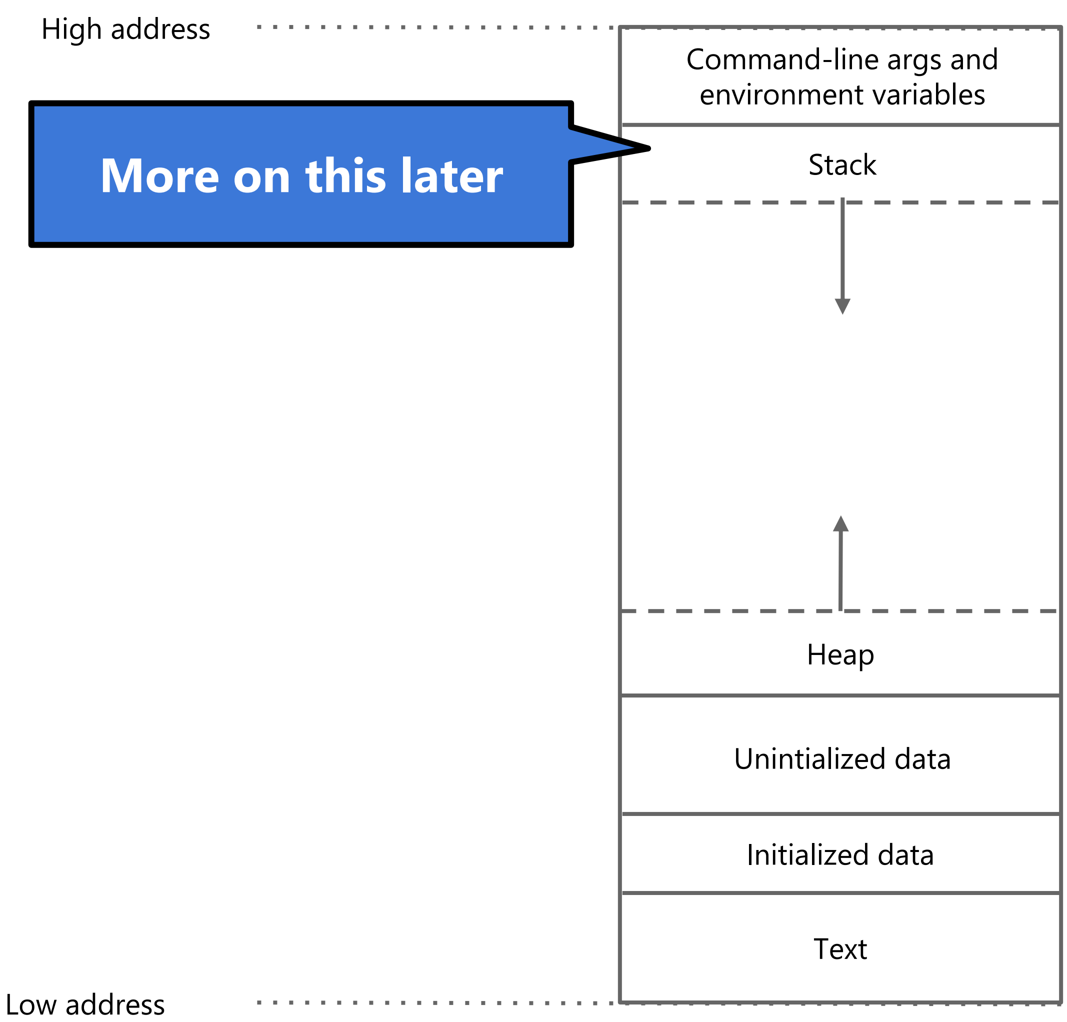
Memory management: program level



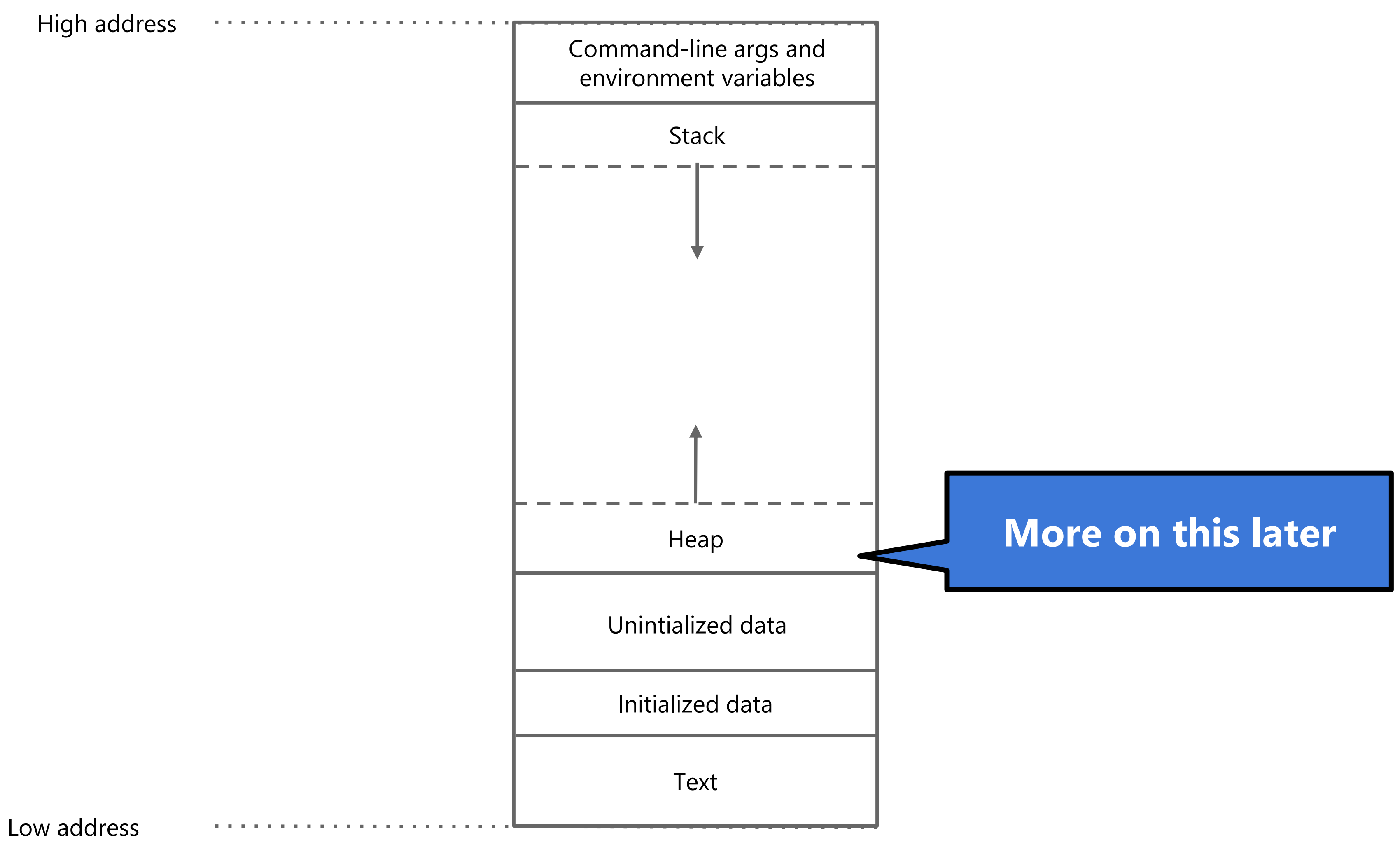
Memory management: program level



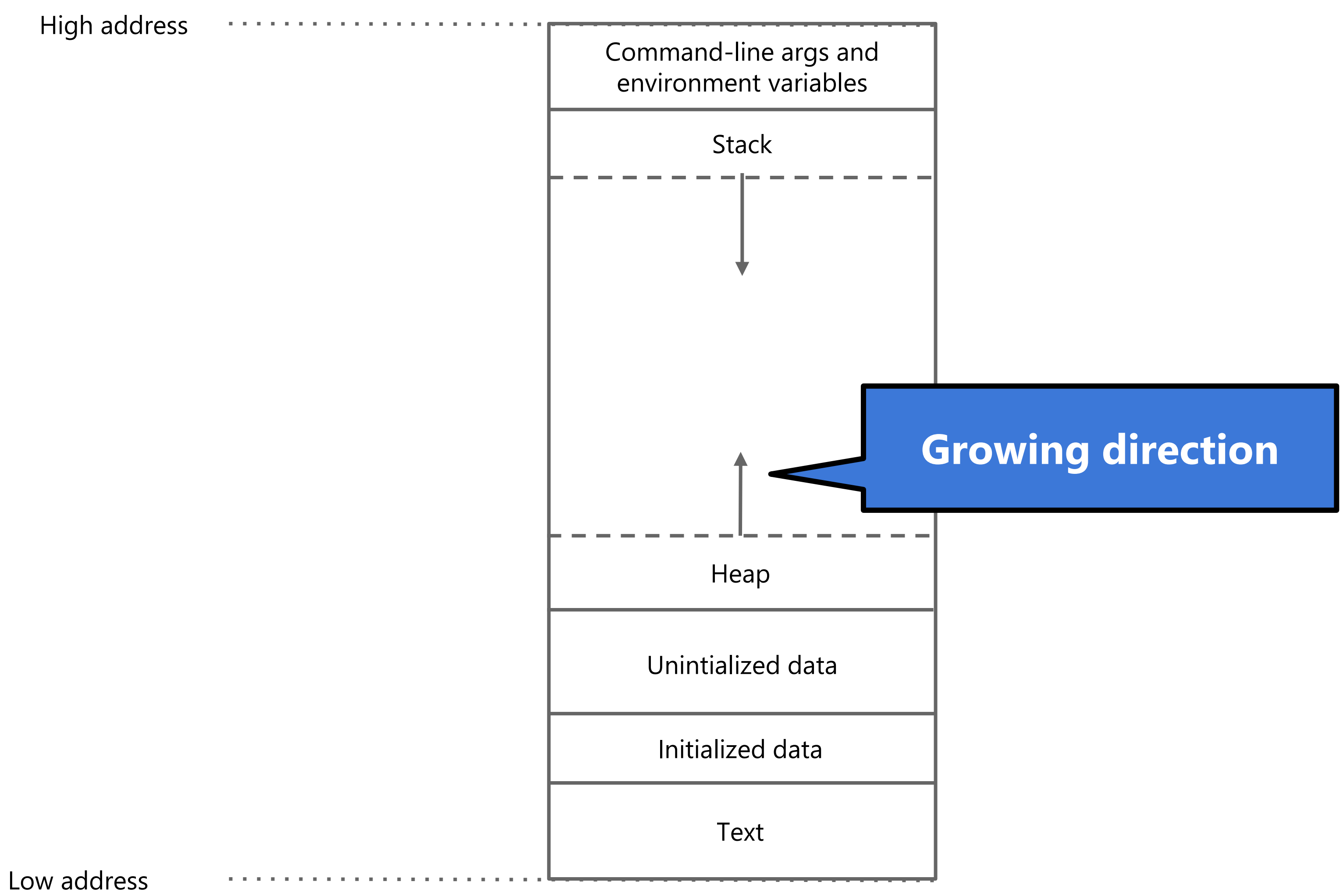
Memory management: program level



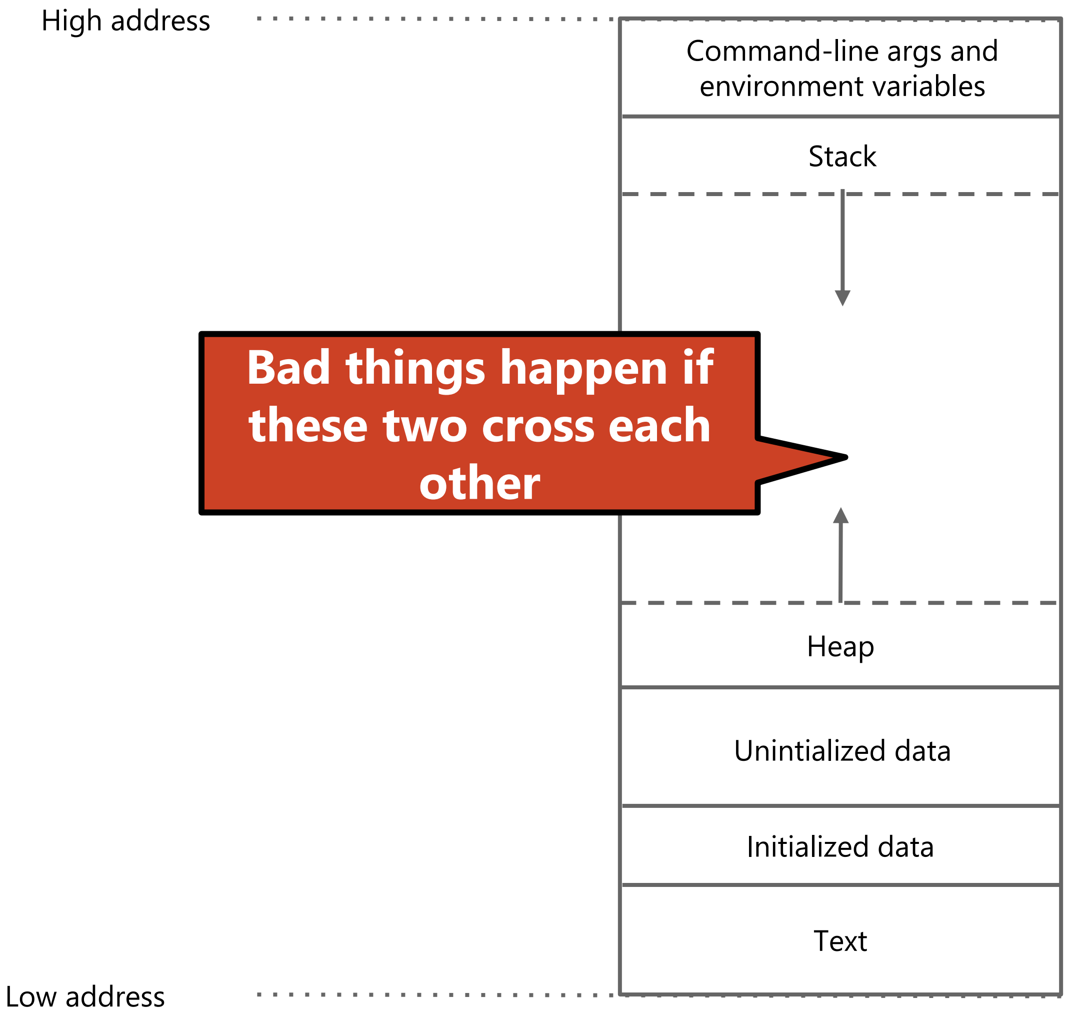
Memory management: program level



Memory management: program level



Memory management: program level



Stack

- ➔ Its a section of the memory layout. Behaves in LIFO just like any given stack
- ➔ It is composed of a set of *stack frames*. Each frame is a **function call**
- ➔ When a **function is called**, a new stack frame is inserted in the stack
- ➔ When a **function returns**, a stack frame gets removed

Stack

➔ Its a section of the memory layout. Behaves in LIFO just like any given stack

➔ It is composed of a set of *stack frames*. Each frame is a **function call**

➔ When a **function is called**, a new stack frame is inserted in the stack

➔ When a **function returns**, a stack frame gets re

What about re
calls?

Stack

- ➔ Each stack frame contains at least the following elements:
  - Storage space for all **automatic variables** for the new called function
  - Line number of the calling function (where to return)
  - Arguments or parameters of the called function

Stack

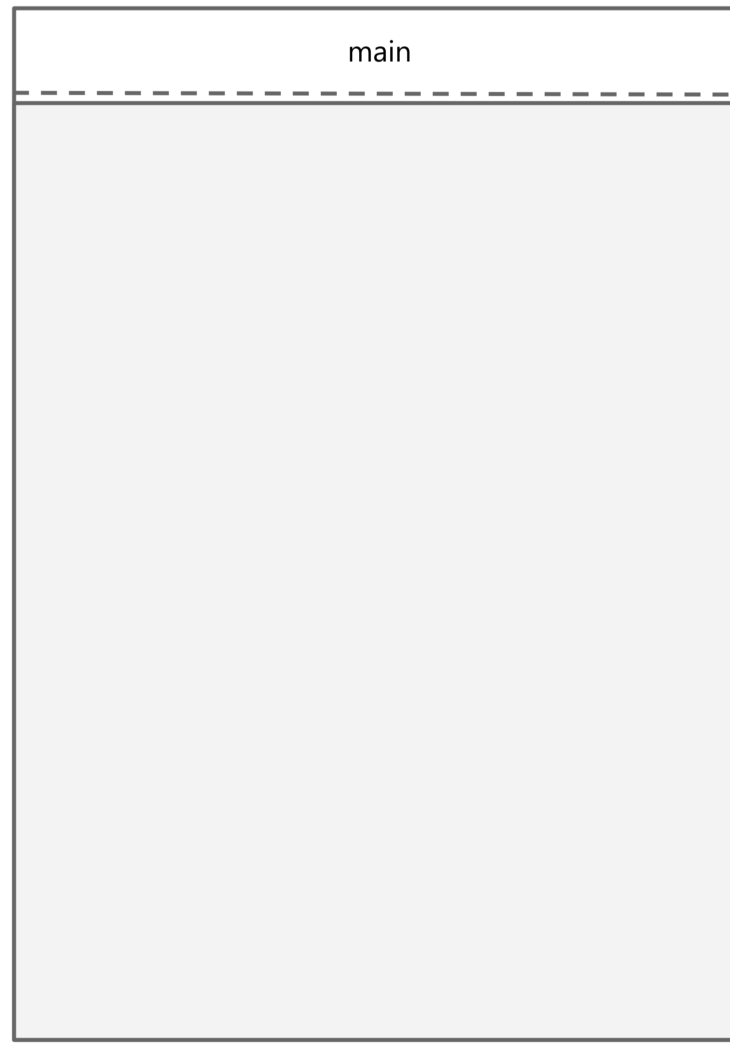
```c
#include <stdio.h>
void first_function(void);
void second_function(int);

int main(void)
{
   printf("hello world\n");
   first_function();
   printf("goodbye goodbye\n");

   return 0;
}


void first_function(void)
{
   int imidate = 3;
   char broiled = 'c';
   void *where_prohibited = NULL;

   second_function(imidate);
   imidate = 10;
}


void second_function(int a)
{
   int b = a;
}
```

Stack

Stack

```c
#include <stdio.h>
void first_function(void);
void second_function(int);

int main(void)
{
   printf("hello world\n");
   first_function();
   printf("goodbye goodbye\n");

   return 0;
}


void first_function(void)
{
   int imidate = 3;
   char broiled = 'c';
   void *where_prohibited = NULL;

   second_function(imidate);
   imidate = 10;
}


void second_function(int a)
{
   int b = a;
}
```

Stack



Stack

```
1   #include <stdio.h>
2   void first_function(void);
3   void second_function(int);
4
5   int main(void)
6   {
7      printf("hello world\n");
8      first_function();
9      printf("goodbye goodbye\n");
10
11     return 0;
12  }
13
14
15  void first_function(void)
16  {
17     int imidate = 3;
18     char broiled = 'c';
19     void *where_prohibited = NULL;
20
21     second_function(imidate);
22     imidate = 10;
23  }
24
25
26  void second_function(int a)
27  {
28     int b = a;
29  }
```

Stack

main

first_function

- ★ Return to **main**, line 9
- ★ Storage for an int
- ★ Storage for a char
- ★ Storage for a void *

Stack

```
1   #include <stdio.h>
2   void first_function(void);
3   void second_function(int);
4
5   int main(void)
6   {
7      printf("hello world\n");
8      first_function();
9      printf("goodbye goodbye\n");
10
11     return 0;
12  }
13
14
15  void first_function(void)
16  {
17     int imidate = 3;
18     char broiled = 'c';
19     void *where_prohibited = NULL;
20
21     second_function(imidate);
22     imidate = 10;
23  }
24
25
26  void second_function(int a)
27  {
28     int b = a;
29  }
```

Stack

main

first_function

- ★ Return to **main**, line 9
- ★ Storage for an int
- ★ Storage for a char
- ★ Storage for a void *

second_function

- ★ Return to **first_function**, line 22
- ★ Storage for an int
- ★ Storage for an int parameter

Stack

```
1   #include <stdio.h>
2   void first_function(void);
3   void second_function(int);
4
5   int main(void)
6   {
7      printf("hello world\n");
8      first_function();
9      printf("goodbye goodbye\n");
10
11     return 0;
12  }
13
14
15  void first_function(void)
16  {
17     int imidate = 3;
18     char broiled = 'c';
19     void *where_prohibited = NULL;
20
21     second_function(imidate);
22     imidate = 10;
23  }
24
25
26  void second_function(int a)
27  {
28     int b = a;
29  }
```

Stack

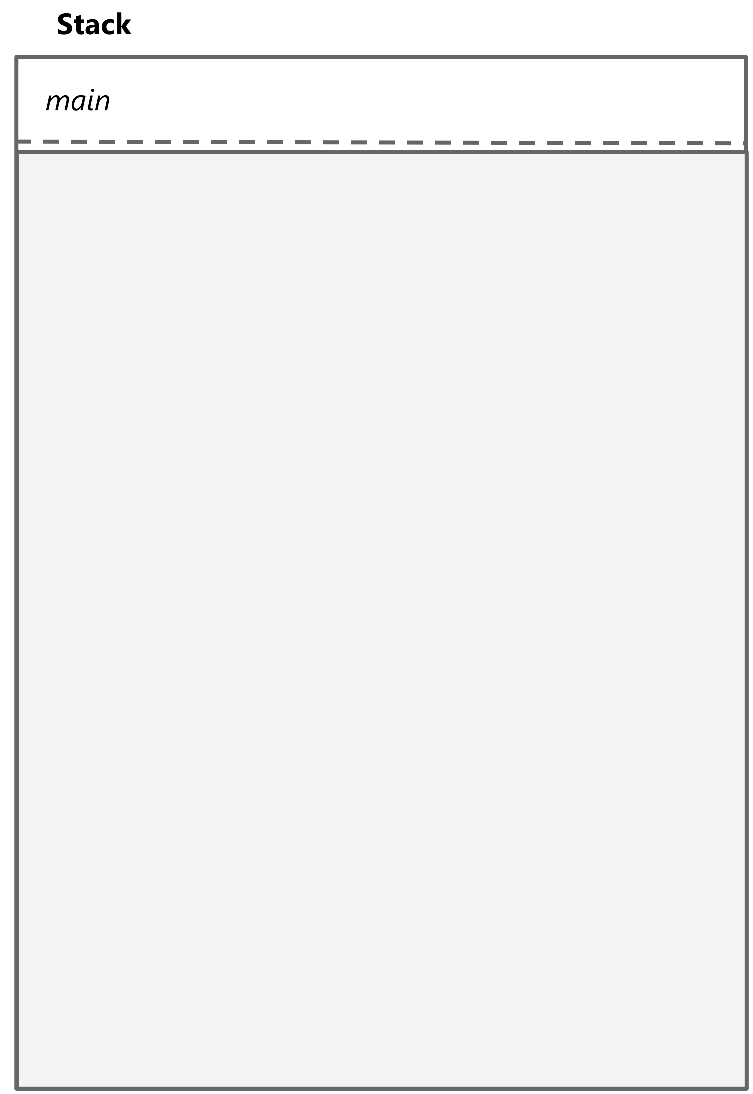
main

first_function

- ★ Return to **main**, line 9
- ★ Storage for an int
- ★ Storage for a char
- ★ Storage for a void *

Stack

```c
#include <stdio.h>
void first_function(void);
void second_function(int);

int main(void)
{
   printf("hello world\n");
   first_function();
   printf("goodbye goodbye\n");

   return 0;
}


void first_function(void)
{
   int imidate = 3;
   char broiled = 'c';
   void *where_prohibited = NULL;

   second_function(imidate);
   imidate = 10;
}


void second_function(int a)
{
   int b = a;
}
```



Stack

- ➔ The **stack is your friend**, it manages the memory for you! It is **transparent** to the programmer!
- ➔ When a stack frame is popped from the stack, all the storage is automatically freed
- ➔ There is a **well-known** limit on variable size
- ➔ Variable scoping can help you understand how the stack works

Heap

- ➔ Section of a process memory layout
- ➔ It is **not managed automatically**, you must be careful when dealing with it.
- ➔ The programmer has to **interact directly** with the heap
  - ◆ Allocate memory
  - ◆ Deallocate/free memory
  - ◆ Resize memory

Heap

- ➔ How does the programmer "talks" with the heap?
  - ◆ The programming language provides an API for this. For example in C you have:
    - malloc
    - calloc
    - realloc
    - free

Heap

→ Can I avoid using the heap?

- Sure you can!
- Some programming languages **don't allow the programmer** to interact with the heap
- The caveat:
  - No interesting data structures!
  - You will be very restricted on what you can do

Heap

```c
#include <stdio.h>

double multiplyByTwo (double input) {
  double twice = input * 2.0;
  return twice;
}

int main (int argc, char *argv[])
{
  int age = 30;
  double salary = 12345.67;
  double myList[3] = {1.2, 2.3, 3.4};

  printf("double your salary is %.3f\n", multiplyByTwo(salary));

  return 0;
}
```

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

double *multiplyByTwo (double *input) {
  double *twice = malloc(sizeof(double));
  *twice = *input * 2.0;
  return twice;
}

int main (int argc, char *argv[])
{
  int *age = malloc(sizeof(int));
  *age = 30;
  double *salary = malloc(sizeof(double));
  *salary = 12345.67;
  double *myList = malloc(3 * sizeof(double));
  myList[0] = 1.2;
  myList[1] = 2.3;
  myList[2] = 3.4;

  double *twiceSalary = multiplyByTwo(salary);

  printf("double your salary is %.3f\n", *twiceSalary);

  free(age);
  free(salary);
  free(myList);
  free(twiceSalary);

  return 0;
}
```

Heap

No using heap

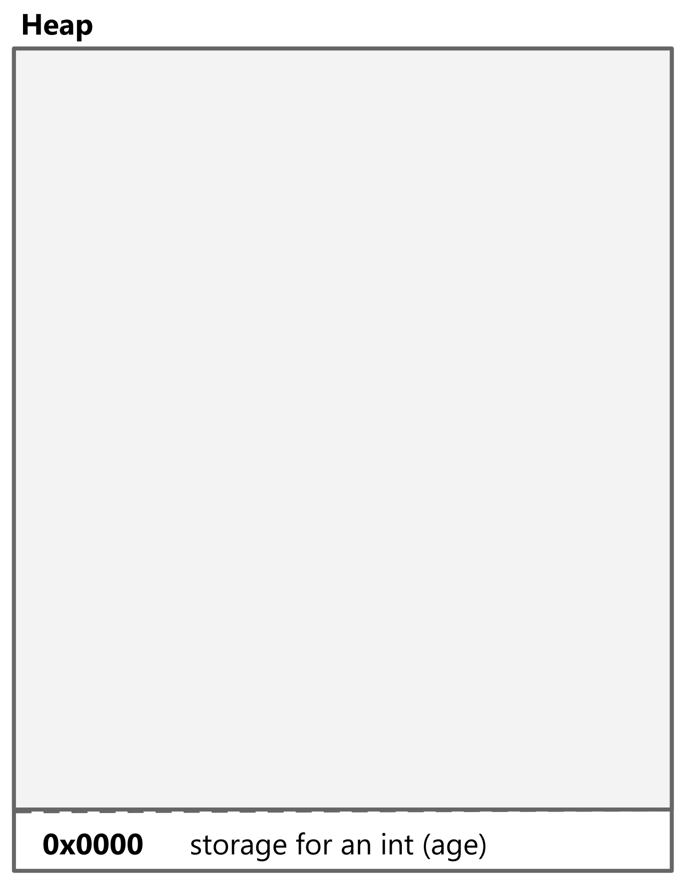
```c
#include <stdio.h>

double multiplyByTwo (double input) {
  double twice = input * 2.0;
  return twice;
}

int main (int argc, char *argv[])
{
  int age = 30;
  double salary = 12345.67;
  double myList[3] = {1.2, 2.3, 3.4};

  printf("double your salary is %.3f\n", multiplyByTwo(salary));

  return 0;
}
```

Using the heap

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

double *multiplyByTwo (double *input) {
  double *twice = malloc(sizeof(double));
  *twice = *input * 2.0;
  return twice;
}

int main (int argc, char *argv[])
{
  int *age = malloc(sizeof(int));
  *age = 30;
  double *salary = malloc(sizeof(double));
  *salary = 12345.67;
  double *myList = malloc(3 * sizeof(double));
  myList[0] = 1.2;
  myList[1] = 2.3;
  myList[2] = 3.4;

  double *twiceSalary = multiplyByTwo(salary);

  printf("double your salary is %.3f\n", *twiceSalary);

  free(age);
  free(salary);
  free(myList);
  free(twiceSalary);

  return 0;
}
```

Heap

```
int main (int argc, char *argv[])
{
  int *age = malloc(sizeof(int));
  *age = 30;
  double *salary = malloc(sizeof(double));
  *salary = 12345.67;
  double *myList = malloc(3 * sizeof(double));
  myList[0] = 1.2;
  myList[1] = 2.3;
  myList[2] = 3.4;

  double *twiceSalary = multiplyByTwo(salary);

  printf("double your salary is %.3f\n",
         *twiceSalary);

  free(age);
  free(salary);
  free(myList);
  free(twiceSalary);

  return 0;
}
```

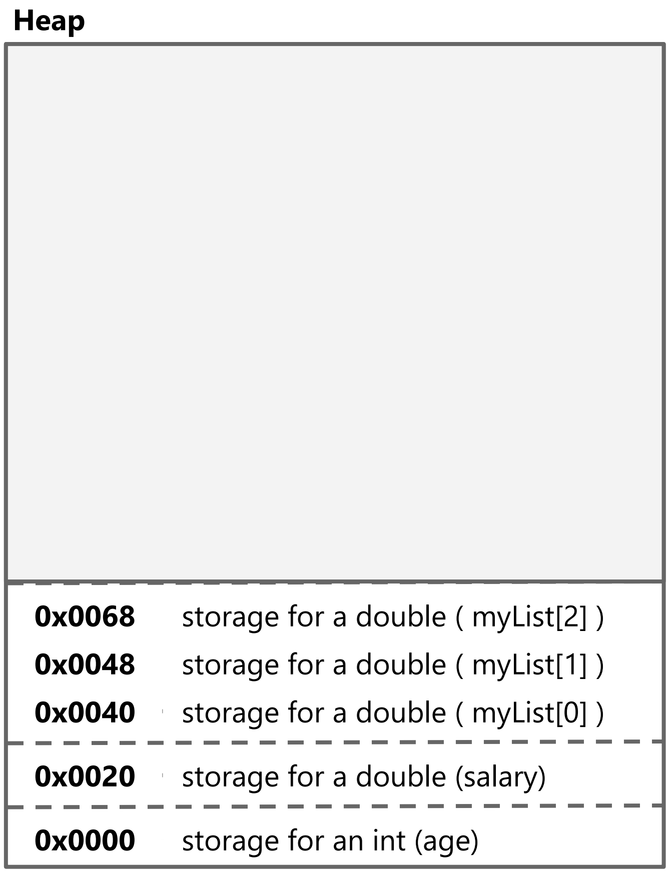


Heap

```c
int main (int argc, char *argv[])
{
  int *age = malloc(sizeof(int));
  *age = 30;
  double *salary = malloc(sizeof(double));
  *salary = 12345.67;
  double *myList = malloc(3 * sizeof(double));
  myList[0] = 1.2;
  myList[1] = 2.3;
  myList[2] = 3.4;

  double *twiceSalary = multiplyByTwo(salary);

  printf("double your salary is %.3f\n",
         *twiceSalary);

  free(age);
  free(salary);
  free(myList);
  free(twiceSalary);

  return 0;
}
```

Heap

```c
int main (int argc, char *argv[])
{
  int *age = malloc(sizeof(int));
  *age = 30;
  double *salary = malloc(sizeof(double));
  *salary = 12345.67;
  double *myList = malloc(3 * sizeof(double));
  myList[0] = 1.2;
  myList[1] = 2.3;
  myList[2] = 3.4;

  double *twiceSalary = multiplyByTwo(salary);

  printf("double your salary is %.3f\n",
         *twiceSalary);

  free(age);
  free(salary);
  free(myList);
  free(twiceSalary);

  return 0;
}
```



Heap

```c
int main (int argc, char *argv[])
{
  int *age = malloc(sizeof(int));
  *age = 30;
  double *salary = malloc(sizeof(double));
  *salary = 12345.67;
  double *myList = malloc(3 * sizeof(double));
  myList[0] = 1.2;
  myList[1] = 2.3;
  myList[2] = 3.4;

  double *twiceSalary = multiplyByTwo(salary);

  printf("double your salary is %.3f\n",
         *twiceSalary);

  free(age);
  free(salary);
  free(myList);
  free(twiceSalary);

  return 0;
}
```

Heap

0x0088	storage for a double (twiceSalary)
0x0068	storage for a double (myList[2])
0x0048	storage for a double (myList[1])
0x0040	storage for a double (myList[0])
0x0020	storage for a double (salary)
0x0000	storage for an int (age)

Heap

```
int main (int argc, char *argv[])
{
  int *age = malloc(size
  *age = 30;
  double *salary = mallo
  *salary = 12345.67;
  double *myList = mallo
  myList[0] = 1.2;
  myList[1] = 2.3;
  myList[2] = 3.4;

  double *twiceSalary = multiplyByTwo(salary);

  printf("double your salary is %.3f\n",
         *twiceSalary);

  free(age);
  free(salary);
  free(myList);
  free(twiceSalary);

  return 0;
}
```

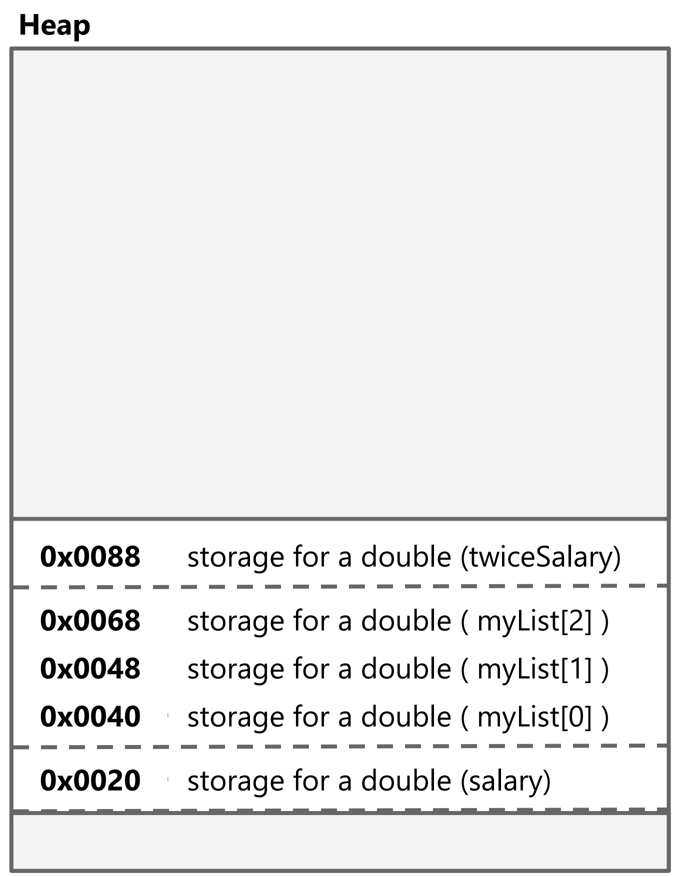
Does malloc under the covers

Heap

0x0088	storage for a double (twiceSalary)
0x0068	storage for a double (myList[2])
0x0048	storage for a double (myList[1])
0x0040	storage for a double (myList[0])
0x0020	storage for a double (salary)
0x0000	storage for an int (age)

Heap

```c
int main (int argc, char *argv[])
{
  int *age = malloc(sizeof(int));
  *age = 30;
  double *salary = malloc(sizeof(double));
  *salary = 12345.67;
  double *myList = malloc(3 * sizeof(double));
  myList[0] = 1.2;
  myList[1] = 2.3;
  myList[2] = 3.4;

  double *twiceSalary = multiplyByTwo(salary);

  printf("double your salary is %.3f\n",
         *twiceSalary);

  free(age);
  free(salary);
  free(myList);
  free(twiceSalary);

  return 0;
}
```



Heap

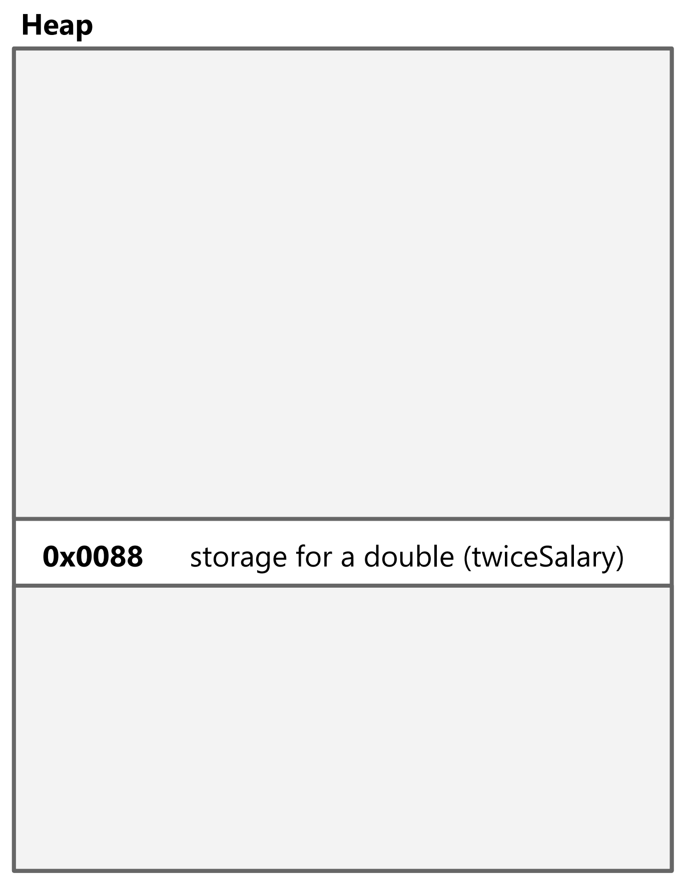
```c
int main (int argc, char *argv[])
{
  int *age = malloc(sizeof(int));
  *age = 30;
  double *salary = malloc(sizeof(double));
  *salary = 12345.67;
  double *myList = malloc(3 * sizeof(double));
  myList[0] = 1.2;
  myList[1] = 2.3;
  myList[2] = 3.4;

  double *twiceSalary = multiplyByTwo(salary);

  printf("double your salary is %.3f\n",
         *twiceSalary);

  free(age);
  free(salary);
  free(myList);
  free(twiceSalary);

  return 0;
}
```

Heap

0x0088	storage for a double (twiceSalary)
0x0068	storage for a double (myList[2])
0x0048	storage for a double (myList[1])
0x0040	storage for a double (myList[0])

Heap

```
int main (int argc, char *argv[])
{
  int *age = malloc(sizeof(int));
  *age = 30;
  double *salary = malloc(sizeof(double));
  *salary = 12345.67;
  double *myList = malloc(3 * sizeof(double));
  myList[0] = 1.2;
  myList[1] = 2.3;
  myList[2] = 3.4;

  double *twiceSalary = multiplyByTwo(salary);

  printf("double your salary is %.3f\n",
         *twiceSalary);

  free(age);
  free(salary);
  free(myList);
  free(twiceSalary);

  return 0;
}
```



Heap

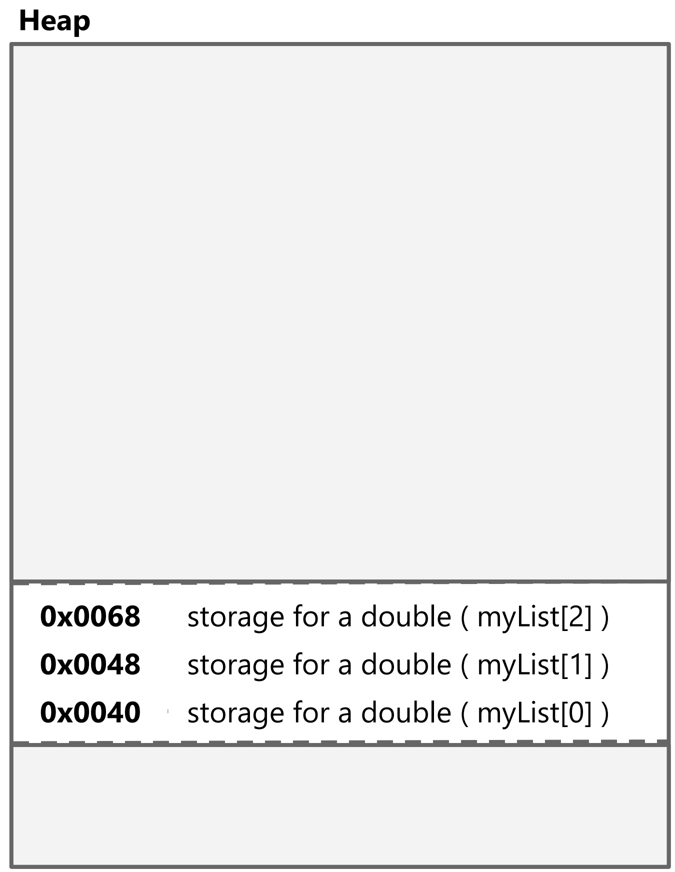
```
int main (int argc, char *argv[])
{
  int *age = malloc(sizeof(int));
  *age = 30;
  double *salary = malloc(sizeof(double));
  *salary = 12345.67;
  double *myList = malloc(3 * sizeof(double));
  myList[0] = 1.2;
  myList[1] = 2.3;
  myList[2] = 3.4;

  double *twiceSalary = multiplyByTwo(salary);

  printf("double your salary is %.3f\n",
         *twiceSalary);

  free(age);
  free(salary);
  free(myList);
  free(twiceSalary);

  return 0;
}
```

Heap

Heap

```
int main (int argc, char *argv[])
{
  int *age = malloc(sizeof(int));
  *age = 30;
  double *salary = malloc(sizeof(double));
  *salary = 12345.67;
  double *myList = malloc(3 * sizeof(double));
  myList[0] = 1.2;
  myList[1] = 2.3;
  myList[2] = 3.4;

  d                                 wo(salary);

  p                                 f\n",
                  alary);

  free(age);
  free(salary);
  //free(myList);
  free(twiceSalary);

  return 0;
}
```

What if I forget to free some storage?



Heap

```
int main (int argc, char *argv[])
{
  int *age = malloc(sizeof(int));
  *age = 30;
  double *salary = malloc
  *salary = 12345.67;
  double *myList = malloc
  myList[0] = 1.2;
  myList[1] = 2.3;
  myList[2] = 3.4;

  d                                   wo(salary);

  p                                   f\n",
                 alary);

  free(age);
  free(salary);
  //free(myList);
  free(twiceSalary);

  return 0;
}
```

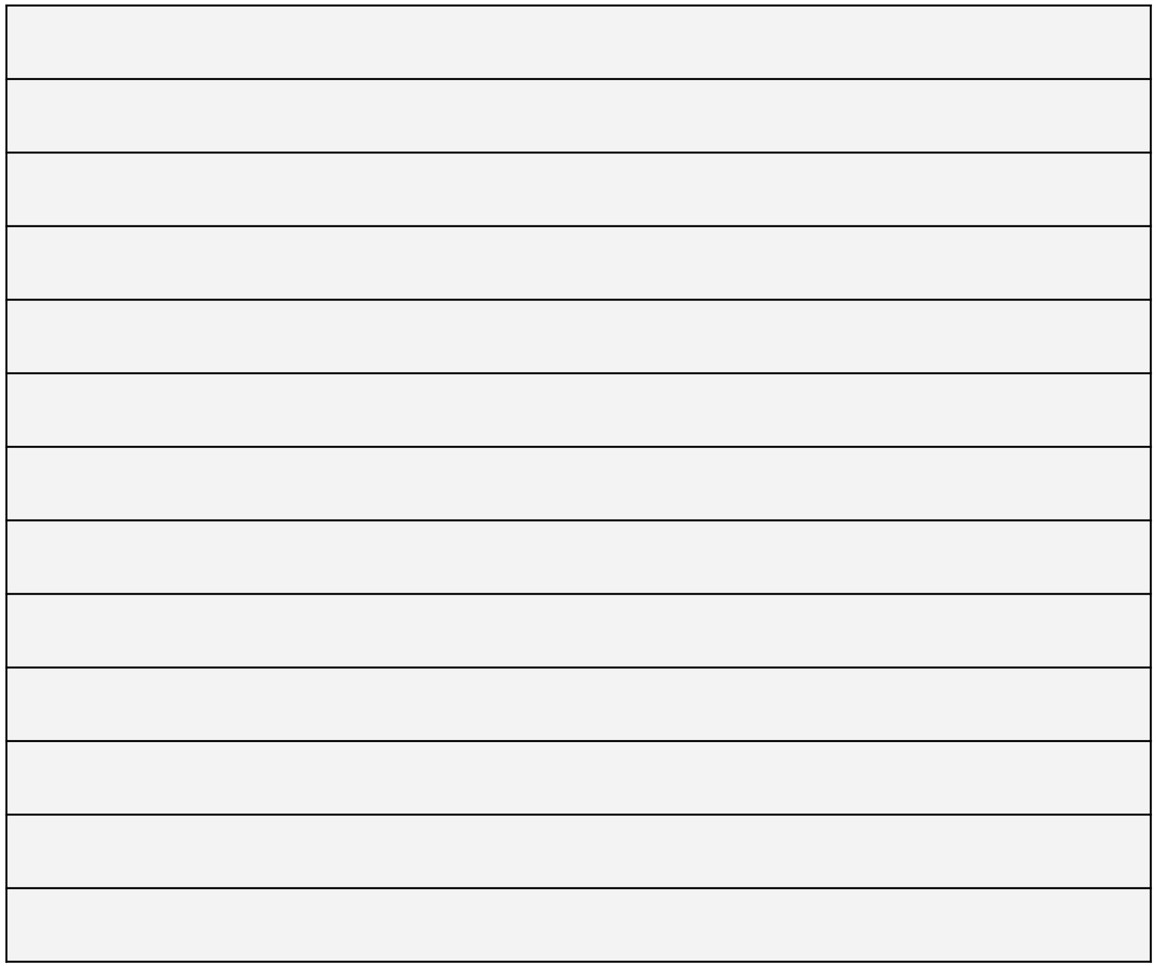
What if I forget to free some storage?

That, my friend, is a memory leak!

Heap

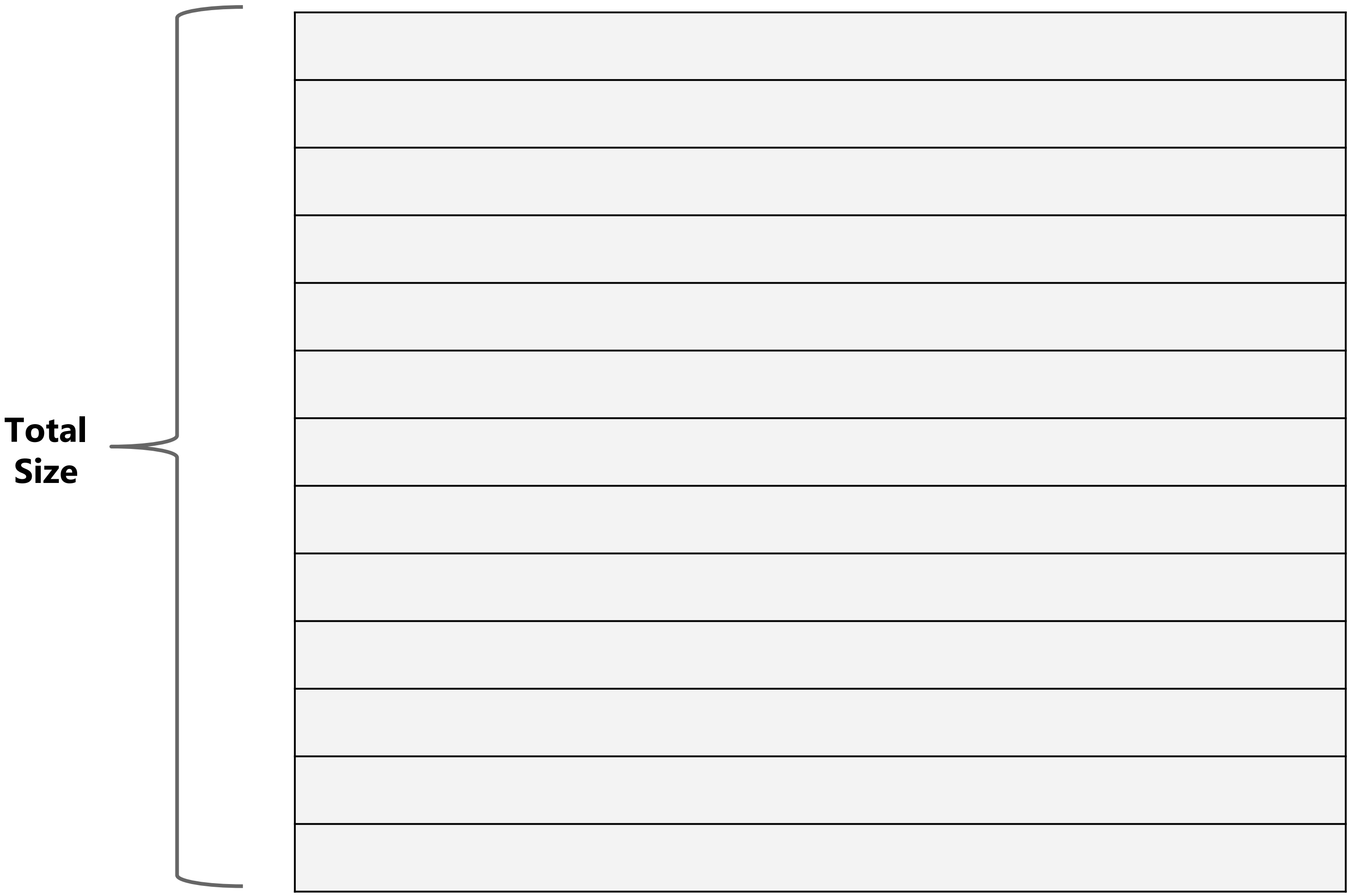
0x0068	storage for a double (myList[2])
0x0048	storage for a double (myList[1])
0x0040	storage for a double (myList[0])

Heap

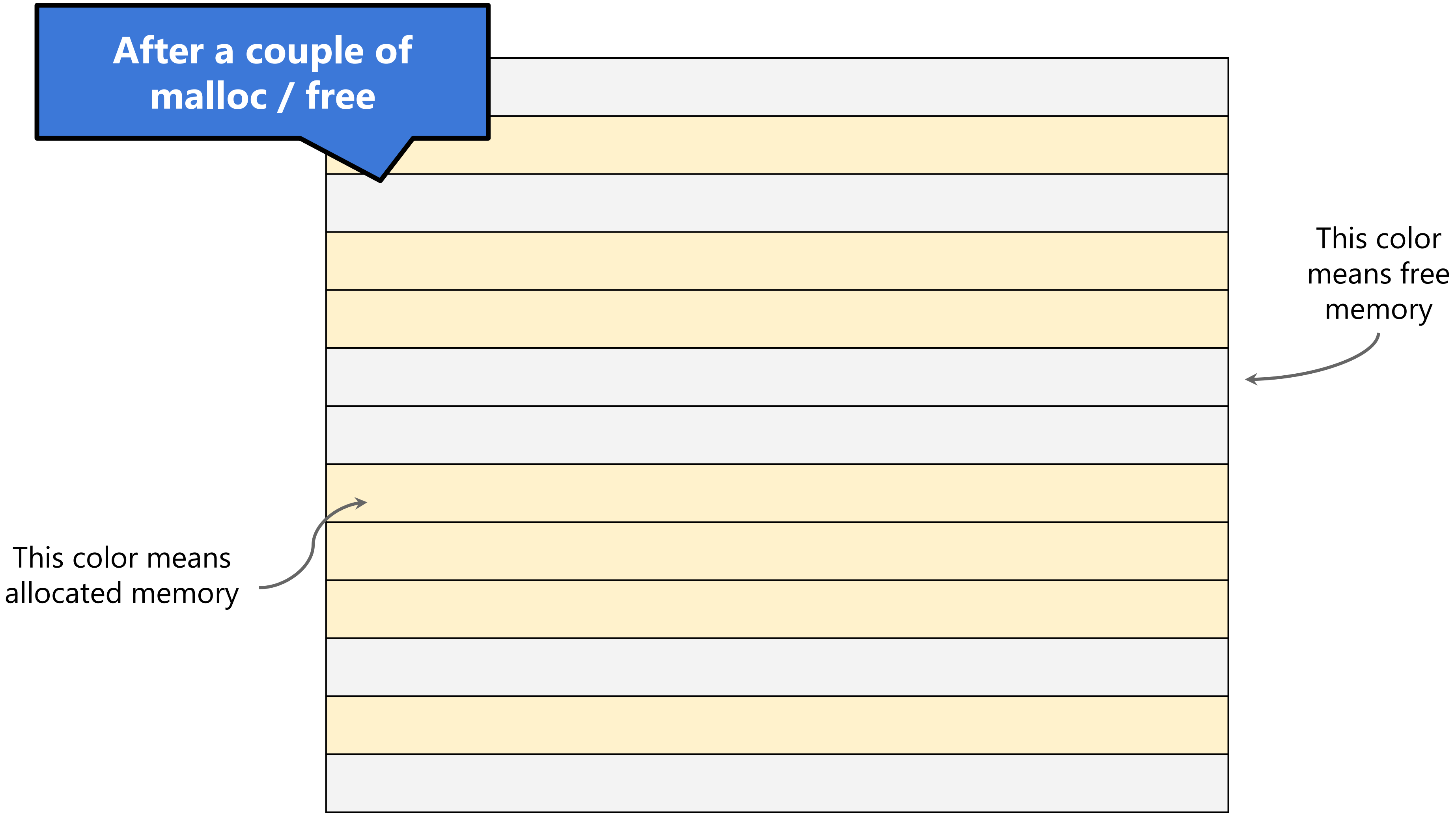


Heap

Heap



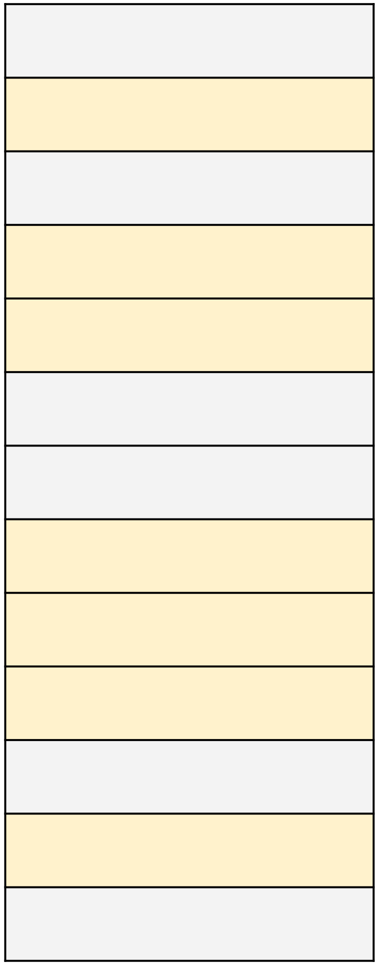
Heap


After a couple of malloc / free
This color means free memory
This color means allocated memory

Heap

- In this scenario, memory becomes fragmented

- What if there's a need for three contiguous blocks of memory?



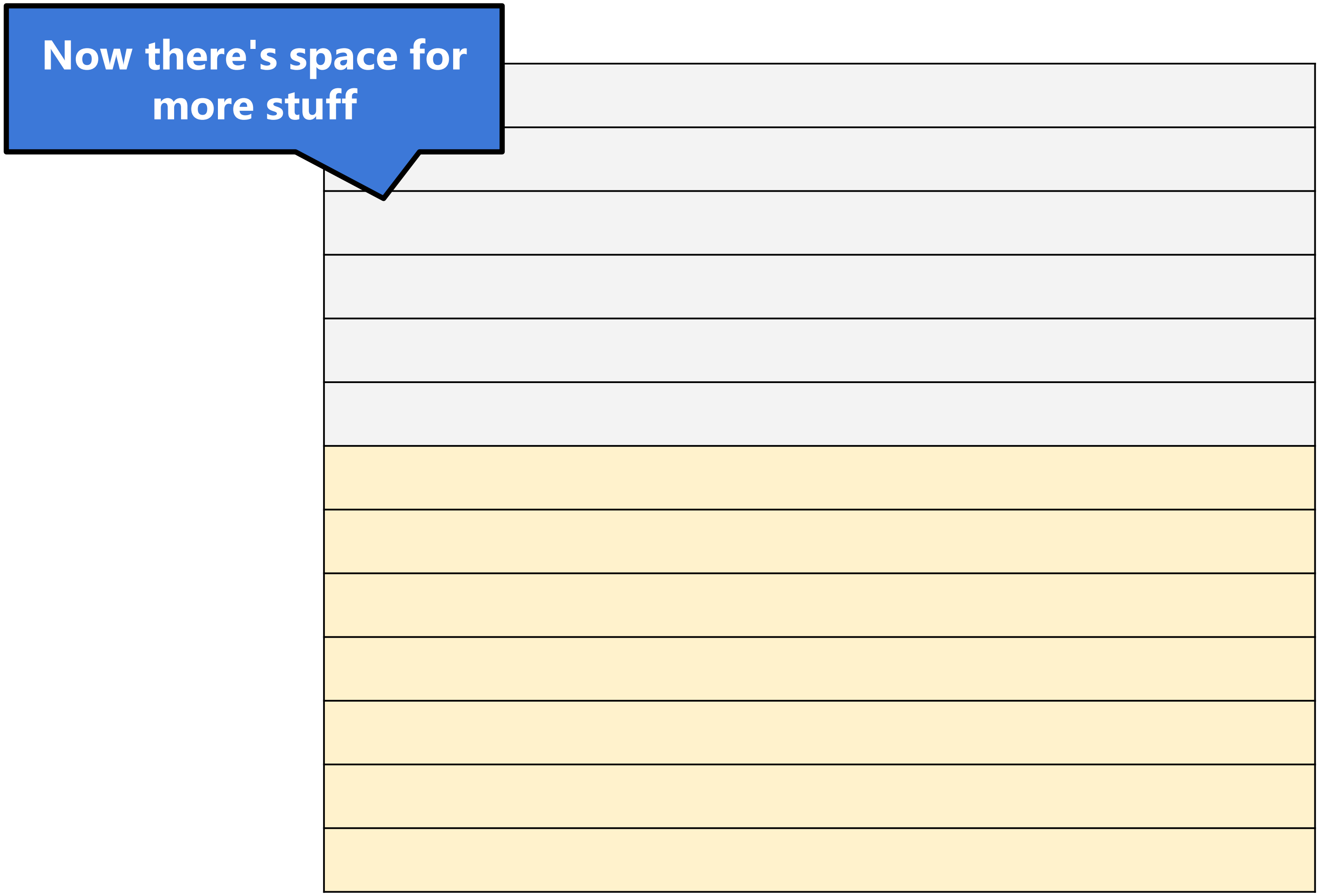
Heap

➔ The operating system "burps" or compacts the memory

➔ Memory "holes" scattered in the memory space are merged

➔ Its an expensive operation

Heap



Heap or stack?

→ Choose the right tool for your needs

→ Use the **heap** if:

- Need to allocate large block of memory (large arrays / structs)
- Need to keep the variable a long time
- Need to create a structure that grows dynamically

Heap or stack?

➔ Use the **stack** if:

- Need only small variables of well-known types
- You know that the variables will not grow dynamically
- Need persist variable only in the scope that they are being used

➔ This decision will be easier in time.

Heap or stack?

	Heap memory	Stack or local memory
+	**Lifetime**: the programmer controls when the memory is allocated and deallocated. Is possible to build data structures and return them to the caller **Size:** the size of the allocated memory can be controlled	**Convenient**: temporary independent memory **Efficient**: time and space efficient **Local copies**: avoid collateral damages
−	**More work:** is the programmer responsibility to handle the memory **More bugs:** the programmer can leave leaks or make mistakes	**Short lifetime**: only exists in a specific scope. The life span is very strict. **Restricted communication**: cannot communicate with the caller from the callee

Memory management in C/C++

➔ What is a **pointer**?

➔ How can I **manipulate memory** in C/C++?

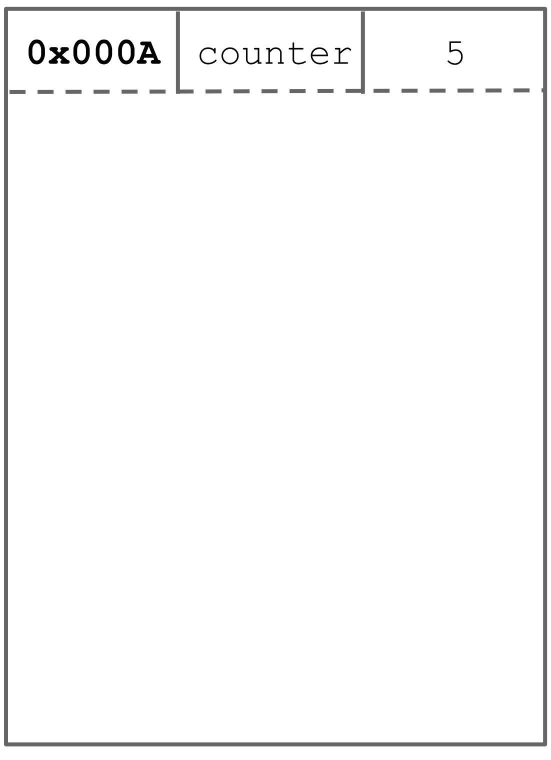
Pointers

- There are many things that can only be done with pointers
- Is one of the most powerful features of C, but one of the most feared
- With pointers the programmer can manipulate memory

Pointers

➔ When a variable is declared, memory is assigned to it.

```
int counter = 5;
```

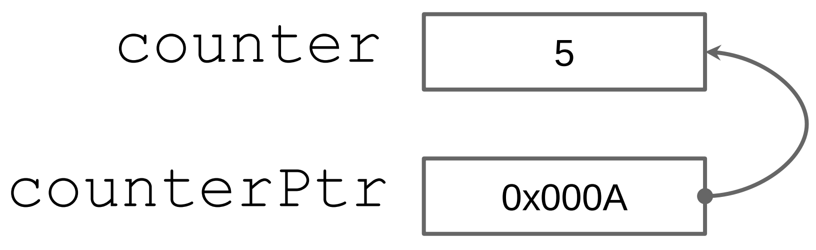


➔ With the variable name is possible to **write** or **read** the value in memory

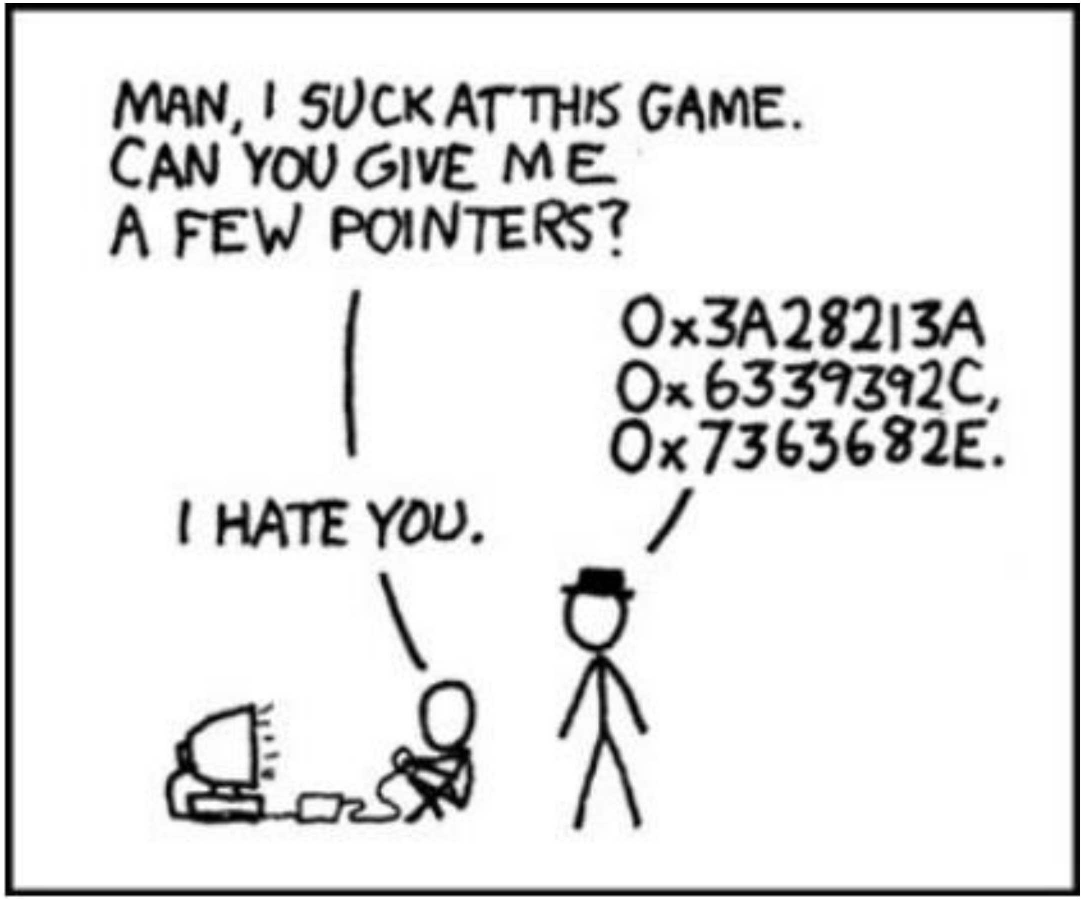
Pointers

➔ A pointer works very different. It doesn't store the value directly, it just points to it.

➔ A pointer **stores a reference** to another value. It stores the memory address of a value in memory



Pointers


MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A 0x6339392C, 0x7363682E.
I HATE YOU.

Pointers

➔ A pointer in C is a **special data type**:

- int*
- double*
- char*
- <struct>*
- void*
- **and many more…**

➔ When a pointer is declared, memory is allocated for it. **But how much memory**?

Pointers

By the way, this is how you declare a pointer…

```
int main (int argc, char *argv[])
{
  int* age;
  char* charPtr = &p;
  void* nothingPtr;

  return 0;
}
```

Pointers

```
int main (int argc, char *argv[])
{
   int* age;
   char* charPtr;
   void* nothingPtr;

   return 0;
}
```

Just like any other variable, pointer need space

This reads: pointer to int, pointer to char, pointer to void

Pointers

```
int main (int argc, char *argv[])
{
  int* age;
  char* charPtr;
  void* nothingPtr;

  return 0;
}
```

Take a guess, how many bytes are assigned to each variable?

Pointers

- ➔ Any pointer variable takes the same amount of space: an **integer**.
- ➔ So, what is the point of specifying a type to the pointer?
- ➔ Specifying a type for a pointer is giving a hint of the type of data that it points to

Pointers

- ➔ Any pointer variable takes the same amount of space: an **integer**.
- ➔ So, what is the point of specifying a type to the pointer?
- ➔ Specifying a type for a pointer is giving a hint of the type of data that it points to

How is this helpful?

Pointers

- ➔ Any pointer variable takes the same amount of space: an **integer**.
- ➔ So, what is the point of specifying a type to the pointer?
- ➔ Specifying a type for a pointer is giving a hint of the type of data that it points to

For error checking

To know how many bytes read later

Pointers

➔ A pointer normally is used to deal with variables in the heap, but it can also be used in the stack

➔ To assign a value to a pointer, you need to use the **& operator** (reference operator)

➔ To get the value pointed by a pointer, use the *** operator**

Pointers

- ➔ A pointer normally is used to deal with variables in the heap, but it can also be used in the stack

What do you think of this?

- ➔ To assign a value to a pointer, you need to use the **& operator** (reference operator)

- ➔ To get the value pointed by a pointer, use the *** operator**

Pointers

- ➔ A pointer normally is used to deal with variables in the heap, but it can also be used in the stack

What do you think of this?

- ➔ To assign a value to a pointer, you need to use the **& operator** (reference operator)
- ➔ To get the value pointed by a pointer, use the *** operator**

Pointers

- ➔ What is the **default value of a pointer** variable?
- ➔ A pointer variable is initialized with a “bad value” a **random address**
- ➔ Always initialize your pointers!

Pointers

- ➔ A pointer can be assigned with the special value NULL
- ➔ The idea is "points to nothing"
- ➔ NULL is equal to 0 **¿Why is this important?**
- ➔ Dereferencing a NULL pointer causes a runtime error

Pointers: The & Operator

➔ There are different ways to set a value to a pointer variable

➔ The most common way is using the & operator

```
int main (int argc, char *argv[])
{
  int data = 55;
  int* dataPtr = &data;

  return 0;
}
```

Pointers: The & Operator

➔ There are different ways to set a value to a pointer variable

➔ The most common way is using the & operator

```
int main (int argc, char *argv[])
{
  int data = 55;
  int* dataPtr = &data;

  return 0;
}
```

Retrieves the address of the data variable and assigns it to dataPtr

Pointers: The * operator

- ➔ How can I get the data pointed by a pointer?

- ➔ Use the ***** **operator**. This unary operator goes on the left of a pointer variable and **retrieves the data** that it points to

- ➔ This operator dereferences a pointer

- ➔ Never dereference a pointer with no value!

Pointers: The * operator

```
int main (int argc, char *argv[])
{
  int data = 55;
  int* dataPtr = &data;

  printf(data);
  printf(dataPtr);
  printf(*dataPtr)

  return 0;
}
```

Pointers: The * operator

```
int main (int argc, char *argv[])
{
  int data = 55;
  int* dataPtr = &data;

  printf(data);
  printf(dataPtr);
  printf(*dataPtr)

  return 0;
}
```

What's displayed on the console?

Pointers: The * operator

```
int main (int argc, char *argv[])
{
  int data = 55;
  int* dataPtr =

  printf(data);
  printf(dataPtr);
  printf(*dataPtr)

  return 0;
}
```

> 55

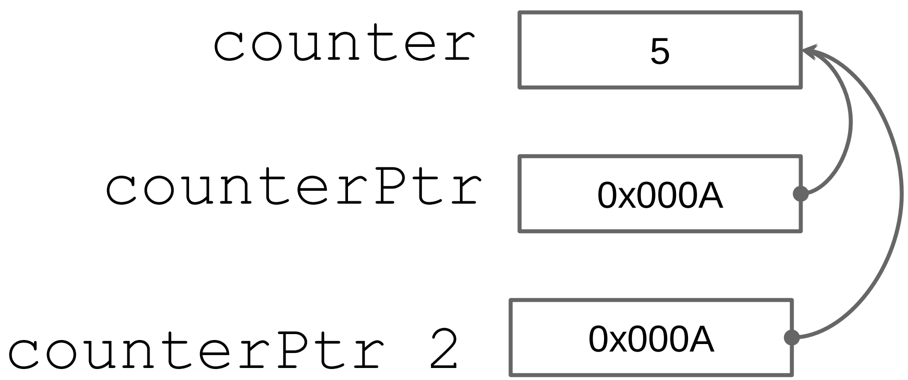
> 0x0032AACC

> 55

Pointers: Sharing

➔ What is sharing?

➔ Having more than one pointers pointing to the same data



Pointers: Sharing

```
int main (int argc, char *argv[])
{
  int data;
  int* dataPtr;
  int* dataPtr_2;

  data       = 256;
  dataPtr    = &data;
  dataPtr_2 = &data;

  return 0;
}
```

Both point to the same data

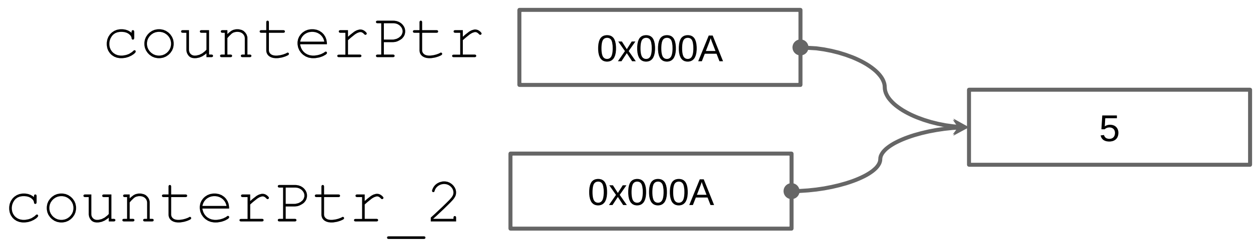
Pointers: Sharing

```
int main (int argc, char *argv[])
{
  int data;
  int* dataPtr;
  int* dataPtr_2;

  data      = 256;
  dataPtr   = &data;
  dataPtr_2 = dataPtr;

  return 0;
}
```

Assigning a pointer to another just copies the address not the data!

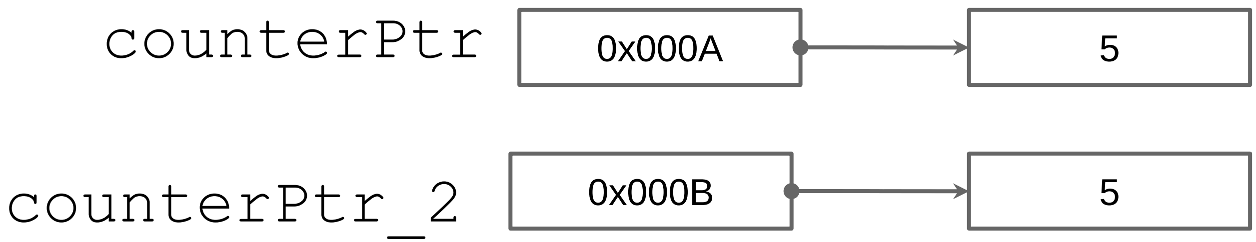
Pointer: Shallow and deep copying

→ **Shallow copy** means just copying the references not the data:



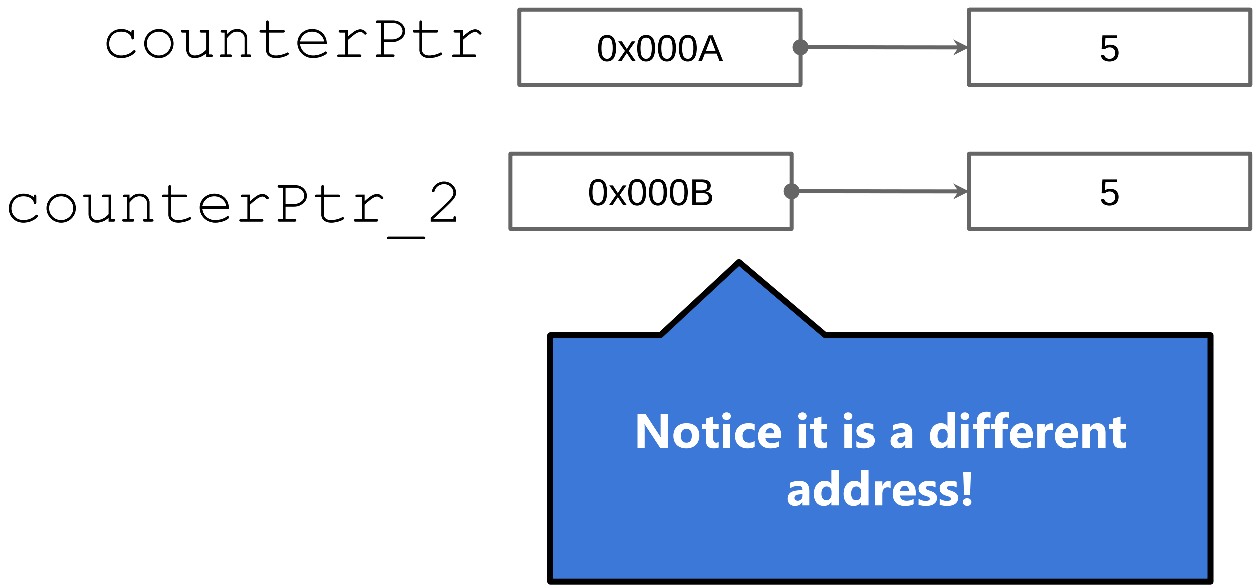
Pointer: Shallow and deep copying

➔ **Deep copy** means also copying the data:



Pointer: Shallow and deep copying

➔ **Deep copy** means also copying the data:



Pointer: Shallow and deep copying

- ➔ To do a deep copy you may need to use the **memcpy, strcpy,** or similar.

- ➔ In some cases it can even need more work, like copying an array or a complex structure

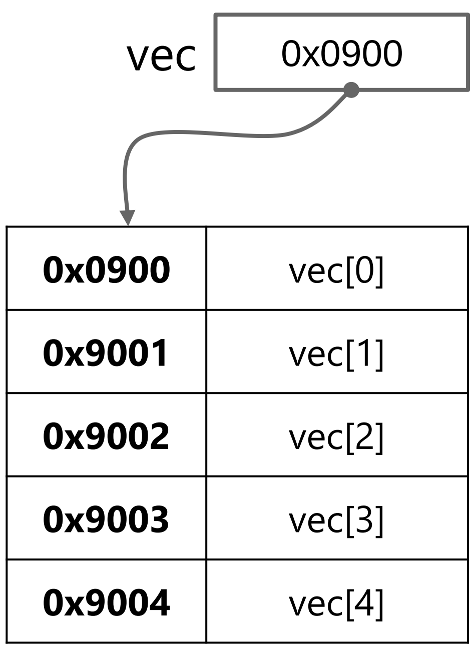
Pointer: Shallow and deep copying

```c
int main (int argc, char *argv[])
{
  int value = 5;
  int *pointer = &value;
  int *pointer2;

  //Shallow copy
  pointer2 = pointer;

  //Deep copy
  pointer2 = malloc(sizeof(int));
  *pointer2 = *pointer;

  return 0;
}
```

Pointers: Arrays and arithmetics

➔ Declaring an array using an expression like *int vec[5];* is allocating a **contiguous block of memory**

➔ The vec variable is a pointer to the first element of the array



Pointers: Arrays and arithmetics

- ➔ Accessing each of the array entry (vec[0], vec[1]...vec[4]) really is pointers arithmetic
  - ◆ Stepping into the array memory *n* number of times

Pointers: Arrays and arithmetic

```
int main(void) {
  int *vec;
  vec = malloc(sizeof(int) * 3);
  vec[0] = 1;
  vec[1] = 2;
  vec[2] = 3;
  printf("vec[2]=%d\n", *(vec+2));
  free(vec);
  return 0;
}
```

Pointers: Arrays and arithmetic

```
int main(void) {
  int *vec;
  int i;
  vec = malloc(sizeof(int) * 3);

  for (i = 0; i < 2; i++) {
    printf(*(vec+i));
  }

  return 0;
}
```

Pointers: examples

```
int main (int argc, char *argv[])
{
  int        data         =        5;
  int      *pointer      =     &data;
  int     *pointer2     =    pointer;
  int       data2        =      data;
  int data3 = *pointer2;
  int **pointerPointer = &pointer;
  data = 8;

  return 0;
}
```

0x9000	5	data
0x9001		
0x9002		
0x9003		
0x9004		
0x9005		
0x9006		
0x9007		
0x9008		

Pointers: examples

```
int main (int argc, char *argv[])
{
  int       data       =       5;
  int     *pointer     =   &data;
  int    *pointer2     = pointer;
  int      data2       =    data;
  int data3 = *pointer2;
  int **pointerPointer = &pointer;
  data = 8;

  return 0;
}
```

0x9000	5	data
0x9001	0x9000	pointer
0x9002		
0x9003		
0x9004		
0x9005		
0x9006		
0x9007		
0x9008		

Pointers: examples

```
int main (int argc, char *argv[])
{
  int         data         =         5;
  int       *pointer      =      &data;
  int      *pointer2      =    pointer;
  int        data2         =      data;
  int data3 = *pointer2;
  int **pointerPointer = &pointer;
  data = 8;

  return 0;
}
```

0x9000	5	data
0x9001	0x9000	pointer
0x9002	0x9000	pointer2
0x9003		
0x9004		
0x9005		
0x9006		
0x9007		
0x9008		

Pointers: examples

```
int main (int argc, char *argv[])
{
  int        data        =        5;
  int       *pointer      =      &data;
  int      *pointer2     =    pointer;
  int        data2        =       data;
  int data3 = *pointer2;
  int **pointerPointer = &pointer;
  data = 8;

  return 0;
}
```

0x9000	5	data
0x9001	0x9000	pointer
0x9002	0x9000	pointer2
0x9003	5	data2
0x9004		
0x9005		
0x9006		
0x9007		
0x9008		

Pointers: examples

```c
int main (int argc, char *argv[])
{
  int        data        =        5;
  int      *pointer      =      &data;
  int     *pointer2      =    pointer;
  int        data2       =       data;
  int data3 = *pointer2;
  int **pointerPointer = &pointer;
  data = 8;

  return 0;
}
```

0x9000	5	data
0x9001	0x9000	pointer
0x9002	0x9000	pointer2
0x9003	5	data2
0x9004	5	data3
0x9005		
0x9006		
0x9007		
0x9008		

Pointers: examples

```
int main (int argc, char *argv[])
{
  int        data        =        5;
  int     *pointer      =     &data;
  int    *pointer2     =    pointer;
  int       data2       =      data;
  int data3 = *pointer2;
  int **pointerPointer = &pointer;
  data = 8;

  return 0;
}
```

0x9000	5	data
0x9001	0x9000	pointer
0x9002	0x9000	pointer2
0x9003	5	data2
0x9004	5	data3
0x9005	0x9001	pointerPointer
0x9006		
0x9007		
0x9008		

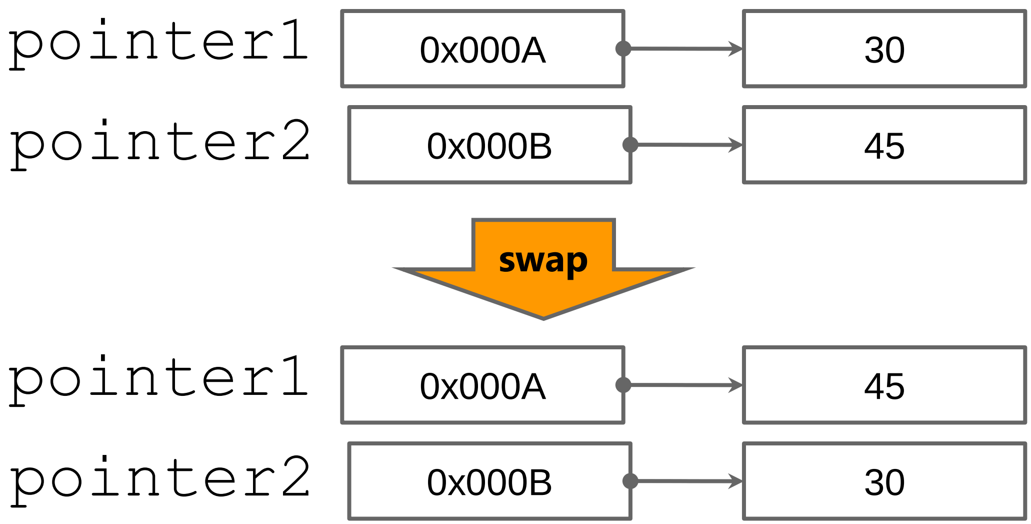
Pointers: examples

```
int main (int argc, char *argv[])
{
  int       data        =        5;
  int      *pointer     =    &data;
  int     *pointer2     =  pointer;
  int       data2       =     data;
  int data3 = *pointer2;
  int **pointerPointer = &pointer;
  data = 8;

  return 0;
}
```

0x9000	8	data
0x9001	0x9000	pointer
0x9002	0x9000	pointer2
0x9003	5	data2
0x9004	5	data3
0x9005	0x9001	pointerPointer
0x9006		
0x9007		
0x9008		

Pointers: examples

```
void swap(int *px, int *py)
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
```



Pointers: examples

```
int* foo()
{
    int temp = 5;
    return &temp;
}
```

What do you think of this code?

References

- ➔ The term **reference** means almost the same thing as the word "pointer"
- ➔ Sometimes pointer is used more in a C/C++ context and reference for other languages
- ➔ Also reference is used more on the context of parameter passing. More on this later.

References

➔ Also C++ supports the reference data type:

```
int main()
{
    int  temp      = 25;
    int& reference = temp;

    printf(reference);

    return 0;
}
```

References

➔ Also C++ supports the reference data type:

```
int main()
{
    int  temp      = 25;
    int& reference = temp;

    printf(reference);

    return 0;
```

Implicit & operation to get the address

An Alias for *temp*

Implicit * operation to get the value pointed

References: Samples

```
void swap (int& first, int& second)
{
  int temp = first;
  first = second;
  second = temp;
}
```

Parameter passing in C/C++

➔ Passing a parameter **by value** means:

- The caller will not see changes made by the callee to the parameter passed
- The callee will receive an independent copy of the parameter

```
void setVar(int x) {
        x = 8;
}

void test() {
    int x = 10;
        setVar(x);
        printf(“Value
%d\n”,x);
}
```

Parameter passing in C/C++

➔ Passing a parameter **by reference** means:

- The callee will receive the reference or pointer of the parameter
- The caller will see changes made by the callee to the parameter passed

```
void setVar(int *xp) {
        *x = 8;
}
void test(){
    int x = 10;
        setVar(&x);
        printf(“Value %d\n”,
x);
}
```

Using the Heap in C/C++

- ➔ In C the main functions to make heap request are **malloc** and **free**
- ➔ In C++ the main functions to make the equivalent requests are **new** and **delete**

Using the Heap in C/C++

- ➔ void* malloc(unsigned long size):
  - ◆ Takes an integer which indicates how many bytes do you want to **allocate**.
  - ◆ **Returns a pointer** to the allocated memory or NULL if the allocation was unsuccessful

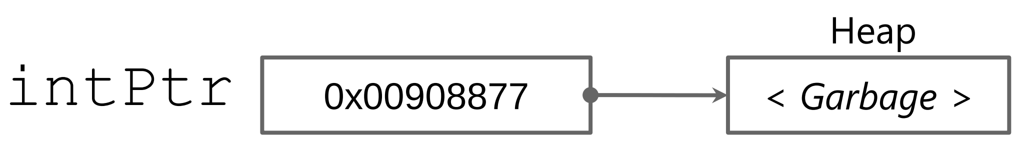
- ➔ void free(void* heapPtr)
  - ◆ Takes a pointer to an allocated memory block on the heap and releases it (**deallocate**).

Using the Heap in C/C++

```
void heapFoo()
{
    int* intPtr;
    intPtr = malloc(sizeof(int));

    *intPtr = 42;
    free(intPtr);
}
```

Using the Heap in C/C++

```
void heapFoo()
{
    int* intPtr;
    intPtr = malloc(sizeof(int));

    *intPtr = 42;
    free(intPtr);
}
```



Using the Heap in C/C++

```
void heapFoo()
{
    int* intPtr;
    intPtr = malloc(sizeof(int));

    *intPtr = 42;
    free(intPtr);
}
```

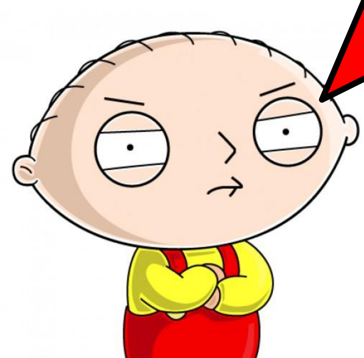
Using the Heap in C/C++

```
void heapFoo()
{
    int* intPtr;
    intPtr = malloc(sizeof(int));

    *intPtr = 42;
    free(intPtr);
}
```

Using the Heap in C/C++

```
void heapFoo()
{
    int* intPtr;
    intPtr = malloc(sizeof(int));

```

intPtr BAD POINT



Using the Heap in C/C++

```
void heapArray() {
   struct fraction* fracts;
   int i;

   fracts = malloc(sizeof(struct fraction) * 100);

   for (i = 0; i < 99; i++) {
      fracts[i].numerator   = 22;
      fracts[i].denominator = 7;
   }

   free(fracts);
}
```

Using the Heap in C/C++

```
void heapArray() {
    struct fraction* fracts;
    int i;

    fracts = malloc(sizeof(struct fraction) * 100);

    for (i = 0; i < 99; i++) {
        fracts[i].numerator   = 22;
        fracts[i].denominator = 7;
    }

    free(fracts);
}
```

Homework: Learn what does the -> operator do

Using the Heap in C/C++

```
char* stringCopy(const char* string) {
    char* newString;
    int length;

    length = strlen(string) + 1;
    newString = malloc(sizeof(char) * length);

    assert(newString != NULL);

    strcpy(newString, string);

    return newString;
}
```

Memory management in Java

➔ Does Java use pointers?

Memory management in Java

- ➔ **Java has pointers** but they are not manipulated with operators as * or &.
- ➔ Simple data types like int, double or char (and others) are the same as in C
- ➔ Objects, strings, arrays (and others) are pointers behind the scenes

Memory management in Java

```
public void JavaShallow() {
    Foo a = new Foo();
    Foo b = new Foo();

    b = a; // Shallow assignment
           // b refers to the same objects as a

    a.Bar();
}
```

Memory management in Java

```
public void JavaShallow() {
    Foo a = new Foo();
    Foo b = new Foo();

    b = a; // Shallow assignment
           // b refers to the same objects as a

    a.Bar();
}
```

Is there any memory leak?

Memory management in Java

➔ The Java approach has two main features:

- **Less bugs**: Java manages the memory for you. The most common bugs related to pointers are gone! This can make the programmer more productive

- **Slower**: Because the language does this work for you it runs slower than C. Most of the time this "speed issue" is irrelevant

Garbage collector

- ➔ Some languages like Java have automatic storage reclamation
  - ◆ The programmer don't need to do explicit deallocation
  - ◆ The block of memory will be abandoned and the garbage collector will release it if not needed anymore
- ➔ The garbage collector keeps a graph/list of all the heap references
  - ◆ When a heap cell is not referenced it will be released

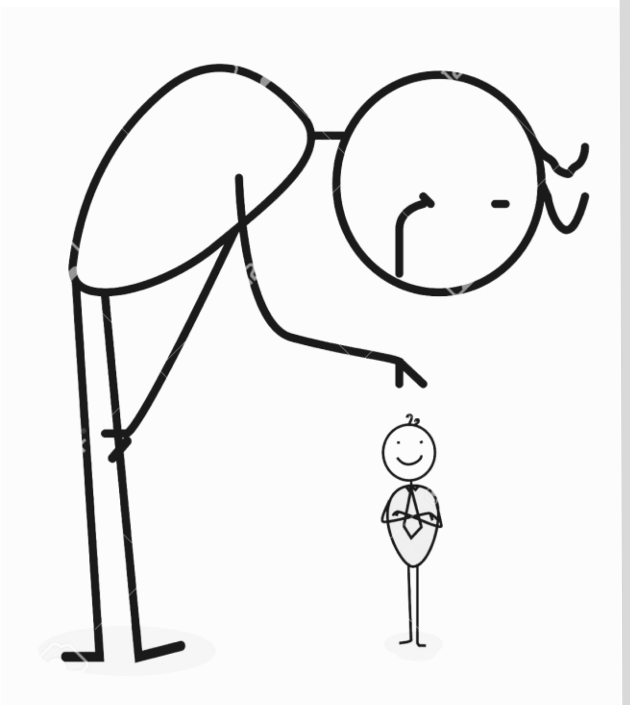
Garbage collector

➔ Usually work in two phases:

- **Marking phase**: identifies all used heap cells. Checks whether a cell is being referenced or not
- **Reclamation phase**: all unmarked cells are returned to the memory pool. To avoid fragmentation, it executes a compaction process

Geek Corner

"Fun" facts for geeks



Today's fact:

Little endian and Big endian

➔ The "endianness" in plain words is how the data is stored in the memory
  - Is how a sequence of bytes are stored in memory

➔ In the **"little endian"** approach, the least significant byte is stored in the smallest address

➔ In the **"big endian"** approach the most significant byte is stored in the smallest address

Today's fact:

Little endian and Big endian

➔ Each memory address can only hold a byte

➔ So for example, if we want to store a 32 bit value, this will be divided in 4 bytes (one byte for each memory address):

90	**AB**	**12**	**CD**

➔ In which order this bytes will be stored? Depending of the endianess

Big endian

90	**AB**	**12**	**CD**
0x1000	0x1001	0x1002	0x1003

Little endian

CD	**12**	**AB**	**90**
0x1000	0x1001	0x1002	0x1003

Geek Corner

Today's fact:

Little endian and Big endian

- ➔ Why this mess happened?!
  - ◆ Back in the old days where the chip technology was very limited, some manufacturers go with one or other endianness for **simplicity**
  - ◆ Old practices still carried today

- ➔ Why this matters to me?
  - ◆ If you exchange data between devices, you may have to do transformations on the bytes, so the receiver can understand the sent message

Geek Corner

Today's fact:

Little endian and Big endian

➔ Is there an easy way to know what kind of endianness has my machine?

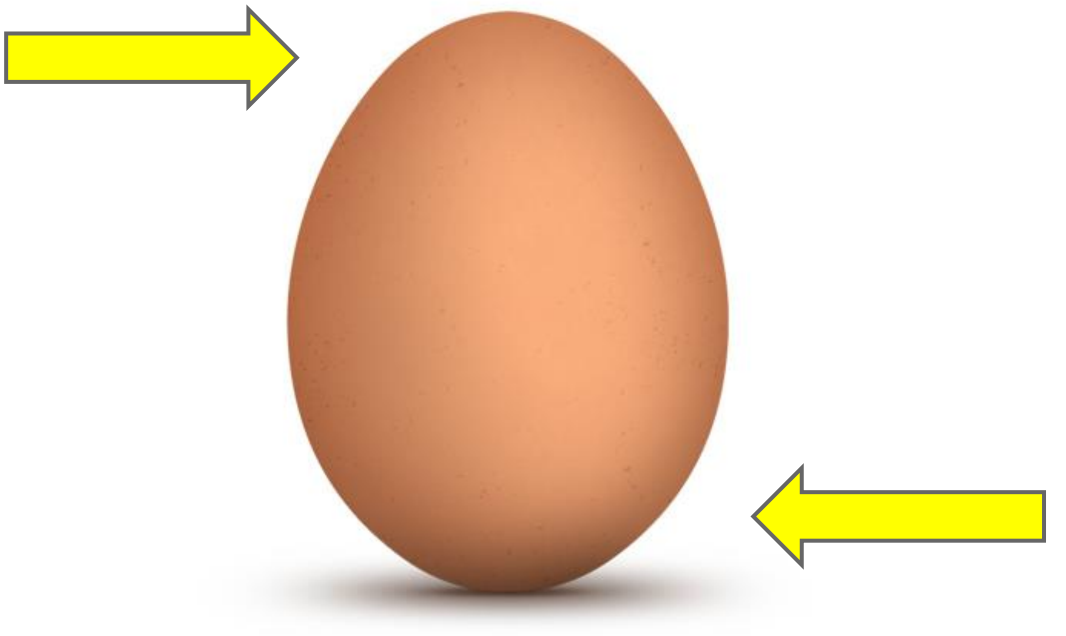
```
#include<iostream>
#include <stdlib.h>
using namespace std;
int main()
{
      unsigned int i = 1;
      char *c = (char*)&i;
      if(*c)
    cout << "Little endian" << endl;
      else
    cout << "Big endian" << endl;
   return 0;
}
```

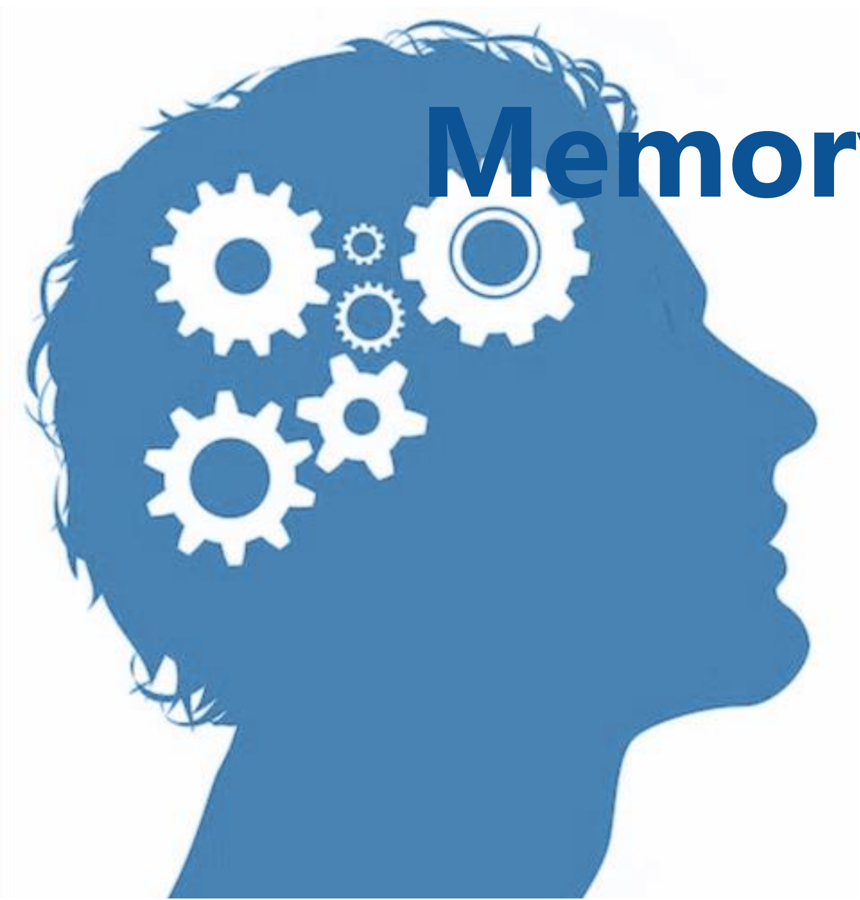
Geek Corner

Today's fact:

Little endian and Big endian

- ➔ Why is called endianness? This term is taken of a satirical novel that mocks of trivial discussions among people such as:
  - ◆ Where to crack an egg? At the small **end**? At the big **end**?





Memory management

CE-2101 Algorithms and Data Structures