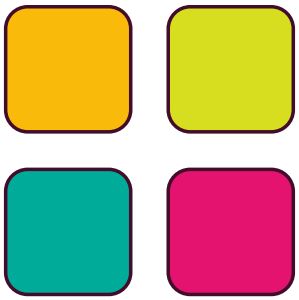
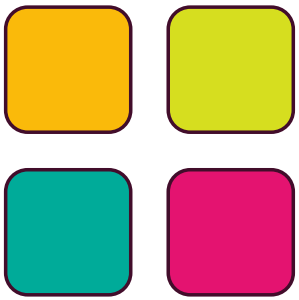


Lecture 6
Introduction to pointers

Gilles Audemard

Ibn Sina School 2015



Introduction

Introduction

- Algorithms consume two kinds of resources : Memory and time
- Two different strategies of memory allocation can be done
  - Static : Allocation is done when the program is launched
  - Dynamic : Allocation is done during the execution, when needed

Static allocation

- Memory allocation is done before the execution :
  - The necessary memory size is known at the compilation stage
  - It is booked in the binary built
  - Memory can be reached at the execution
- During the execution, no allocation is performed
- More efficient (dynamic allocation is a costly operation)

Dynamic allocation

- Mostly programs need variable memory resources
- It is necessary to ask (at arbitrary point of the execution) to the system new memory areas
- It is necessary to free dynamic allocations when they become useless
  - The programer has to do that ?
  - The system has to do that (garbage collector) ?

Potential problems

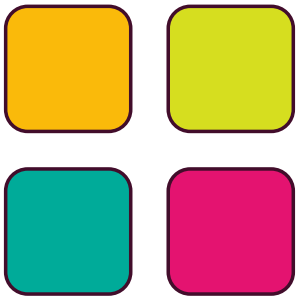
- Indexes out of bounds (in array)
- References to a non allocated pointer
- Memory leaks

C - C++

- No check
- No garbage collector
- Programmer is considered responsible

Java - Python. . .

- Check
- Garbage collector
- Programmer is considered irresponsible



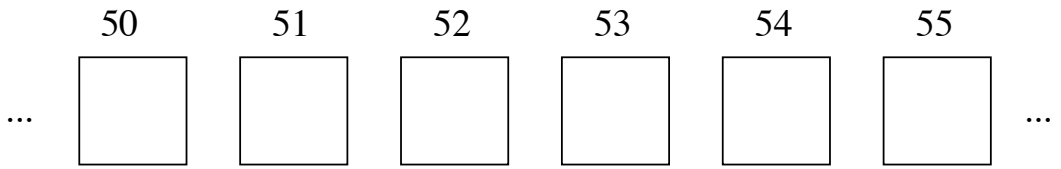
Memory representation

Introduction

- Each byte can be characterized by its address
- A variable uses memory space to store its content
- A variable has thus an address in memory
- This address is called pointer to its content

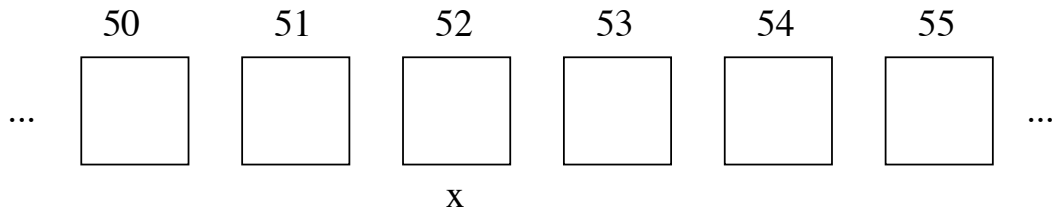
Introduction

- Each byte can be characterized by its address
- A variable uses memory space to store its content
- A variable has thus an address in memory
- This address is called pointer to its content



Introduction

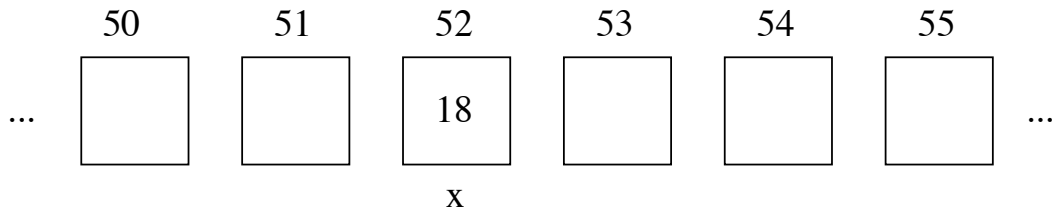
- Each byte can be characterized by its address
- A variable uses memory space to store its content
- A variable has thus an address in memory
- This address is called pointer to its content




```
int x;
```

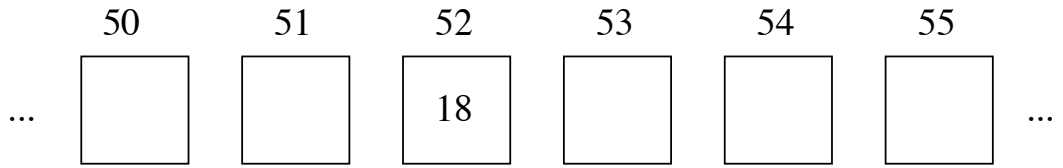
Introduction

- Each byte can be characterized by its address
- A variable uses memory space to store its content
- A variable has thus an address in memory
- This address is called pointer to its content




```
int x;
x = 18;
```

Address of variable




```
int x;
x = 18;
```

- How to know the address where `x` is stored ?
- This is done with the symbol `&` : `&x`

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

int main() {
  int x = 18;
  printf("%d\n",x);
  printf("%p\n",&x);
}
```

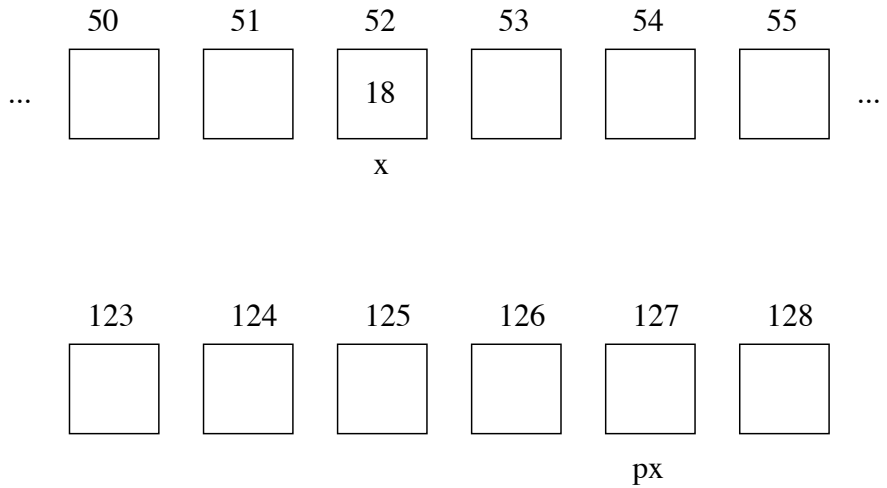
examples/alloc1.c

Pointer variable

- It is possible to create variables that have the type pointer
- They are intended to store memory address
- This is done with the symbol $\star$

Pointer variable

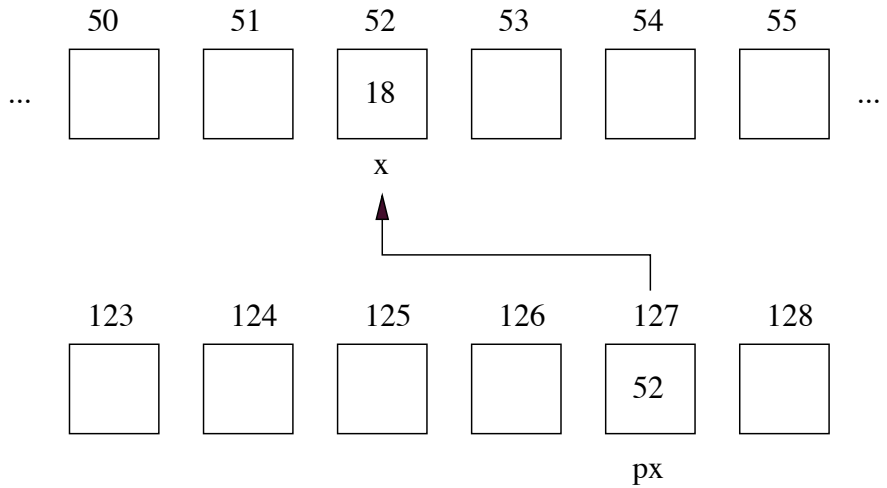
- It is possible to create variables that have the type pointer
- They are intended to store memory address
- This is done with the symbol $\star$




```
int x=18;
int *px = NULL;
```

Pointer variable

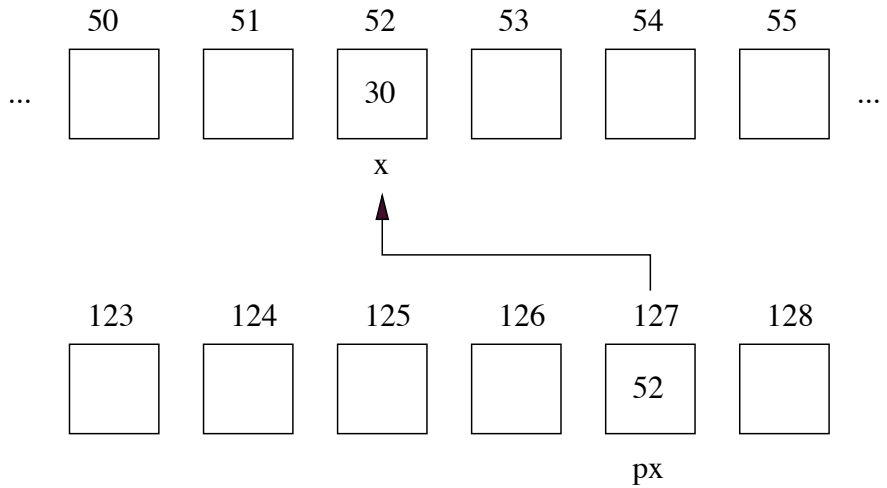
- It is possible to create variables that have the type pointer
- They are intended to store memory address
- This is done with the symbol $\star$




```
int x=18;
int *px = NULL;
px = &x;
```

Pointer variable

- It is possible to create variables that have the type pointer
- They are intended to store memory address
- This is done with the symbol $\star$




```
int x=18;
int *px = NULL;
px = &x;

*px = 30;
```

The * operator

The * operator is used for declaration and use ! !

- Two distinct usages that must not confuse you
- You can use it at the left or right of an assignment
- This operator returns the object or its value that starts at this memory address

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

int main() {
  int x = 18;
  int *px = &x;
  printf("%d\n",x);
  printf("%d\n",*px);
  printf("%p\n",&x);
  printf("%p\n",px);
}
```

examples/alloc2.c

Always assign the `NULL` constant to pointers that are not initialized

Our first Seg fault !

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

int main() {
  int *px = NULL;
  printf("%p\n",px);
  printf("%d\n",*px);
}
```

examples/alloc3.c

- What's happen ?
- The program try to access in a forbidden area of the memory ! !
- The program has only the right to access on a dedicated area
- Each time, one try to access/write elsewhere a segmentation fault is done

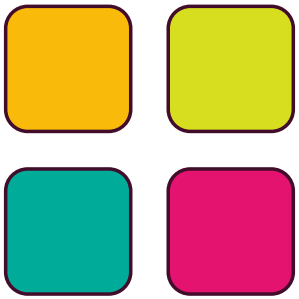
Pointers and types

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

int main() {
  int x = 18;
  int *px = &x;
  printf("%d\n",x);
  printf("%d\n",*px);
  printf("%p\n",&x);
  printf("%p\n",px);
}
```

examples/alloc2.c

- Be careful : a pointer on a variable with type `A` must be of type `*A` !
- Otherwise you obtain this warning

  `warning: initialization from incompatible pointer type`
- It is important to read warning (and of course error) during the compilation phase
- This warning is important, we will see why later in the week

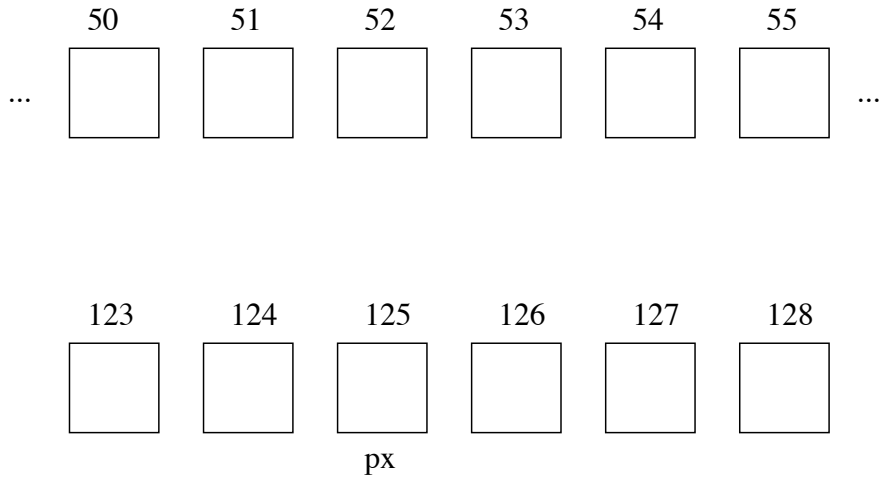


Dynamic allocation

Introduction

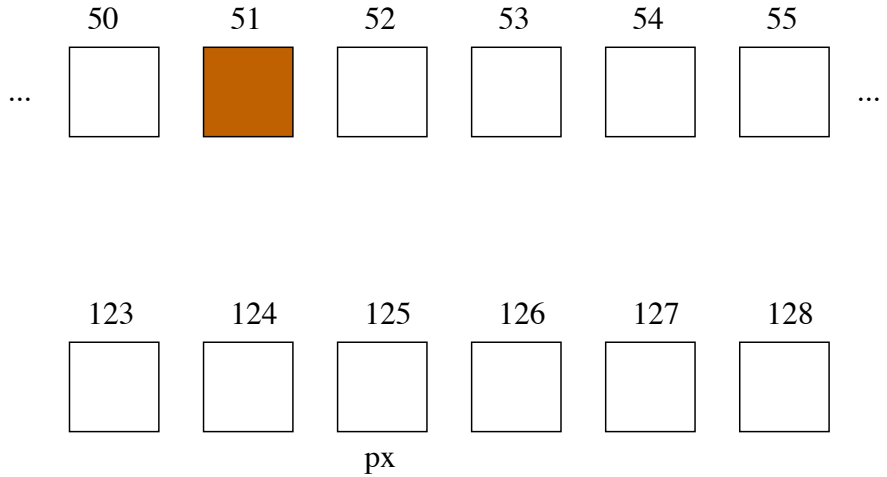
- Until now, all variables are statically allocated
- Suppose we have a pointer `px` of type int that not point on a variable
- Thus, we can not write `*px = 3`
- `px` does not point on a declared variable
- It is possible to allocate space for `px`

A first look of dynamic allocation



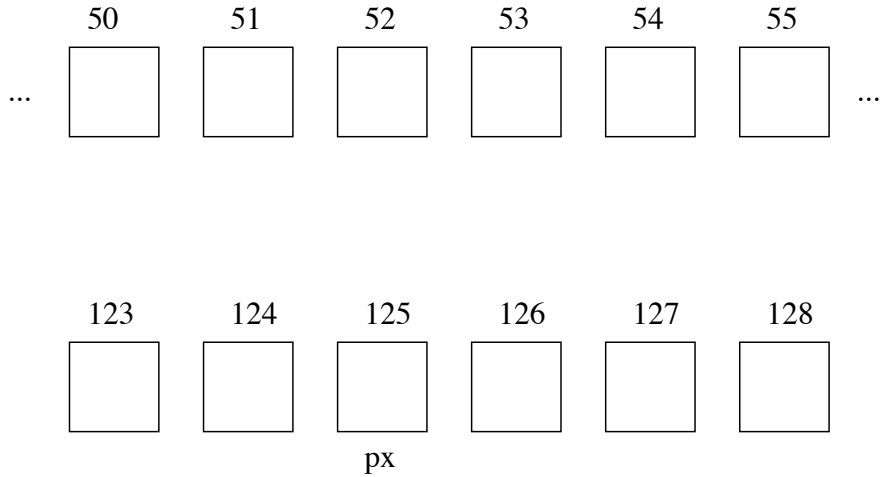

```
int *px;
```

A first look of dynamic allocation




```
int *px;
px = (int*)malloc(sizeof(int))
```

A first look of dynamic allocation




```
int *px;
px = (int*)malloc(sizeof(int))
```

Example

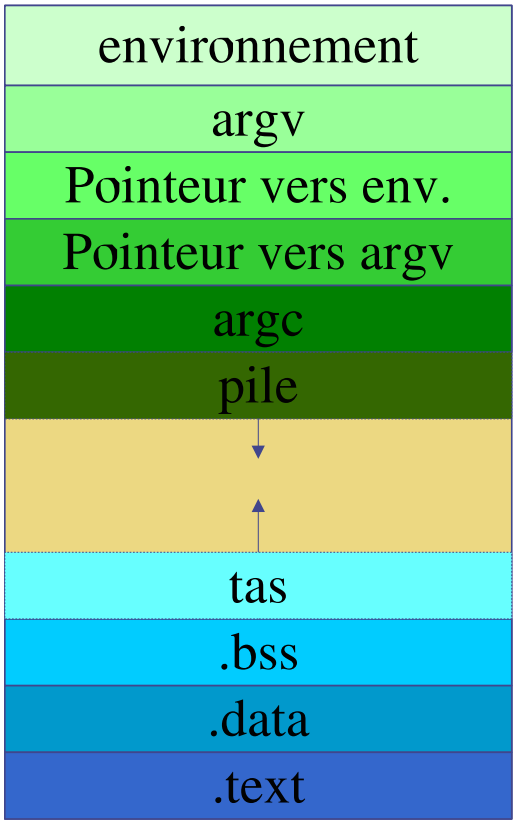
```c
#include<stdlib.h>
#include<stdio.h>

int main() {
  int *p = NULL;
  printf("address = %p\n",p); // 0x0 (NULL)
  p = (int*) malloc(sizeof(int));
  printf("address = %p\n",p); // an address
  *p = 12;  // We store 12 in the reserved area
}
```

examples/alloc4.c

Allocation functions

- `void *malloc(size_t size)`
  - Allocation of `size` bytes of memory
  - Cast is necessary
  - No init !!
- `void *calloc(size_t nb,size_t size)`
  - Allocation of `nb` elements of `size` bytes
  - Cast is necessary
  - Initialisation

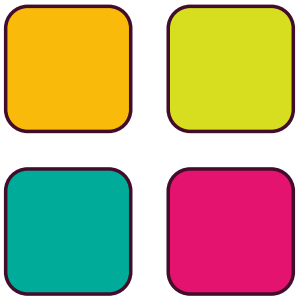
Memory organisation



- Memory is organized in several parts
  - `.text` contains instructions : read-only access
  - `.data` stores global datas initialized
  - `.bss` store global variables non initialized
  - User stack frame contains the stack and the heap : used for local and dynamic variables

Stack and heap

- The stack
  - Stored in the high part of the memory
  - Increase with decreasing addresses
  - LIFO (see course of L. Simon)
  - Used for function calls : parameters, registers, local variables
- The heap
  - Increase with increasing variables
  - Huge datas are stored inside
  - Dynamic allocations also



Memory Leaks

The rule !

All dynamic allocations have to be free

- It is essential
- However, memory leaks can appear
- `void free(void *ptr)`

Example

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

int main() {
  int *p;
  p = (int*) malloc(sizeof(int));

  // perform whatever you want

  free(p); // when p becomes useless

}
```

examples/alloc5.c

Example of memory leaks

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

int main() {
  long *p;
  long i;
  for(i = 0; i <10000000000;i++) {
    p = (long*) malloc(sizeof(long));
    // perform whatever you want
    //    free(p);
  }
}
```

examples/alloc6.c

- Each step in the loop contains a memory leak
- At the end, 8×10^9 bytes are lost !