Introduction to Programming

Lecture 5 — Computers' memory

Nicolas Gry

Université de Saint-Étienne | Year 2026/2027

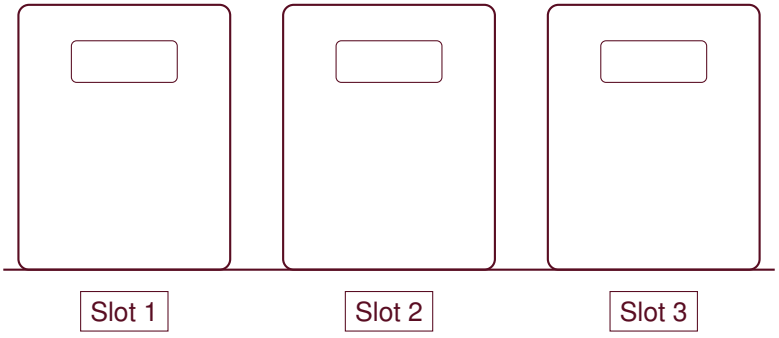
digicrea digital creativity
Université Jean Monnet Saint-Étienne

-1 Recap’ time

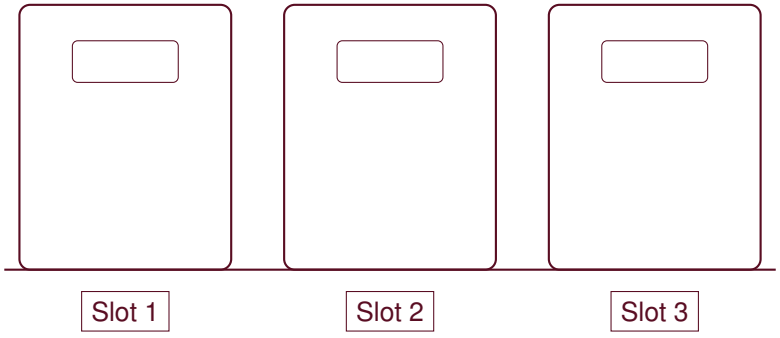
00 Assignment review

01 Memory spots, addresses and pointers

Boxes in your memory

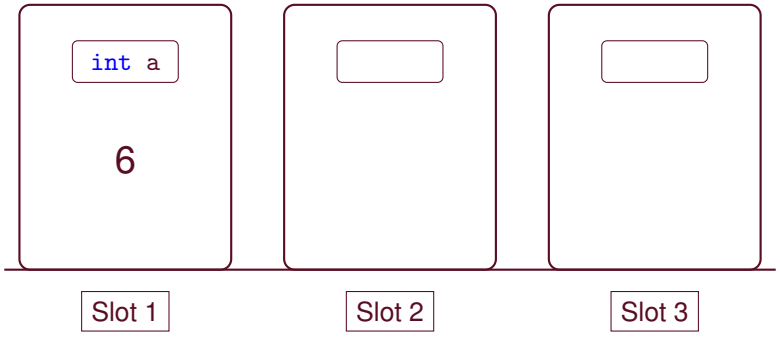


Boxes in your memory



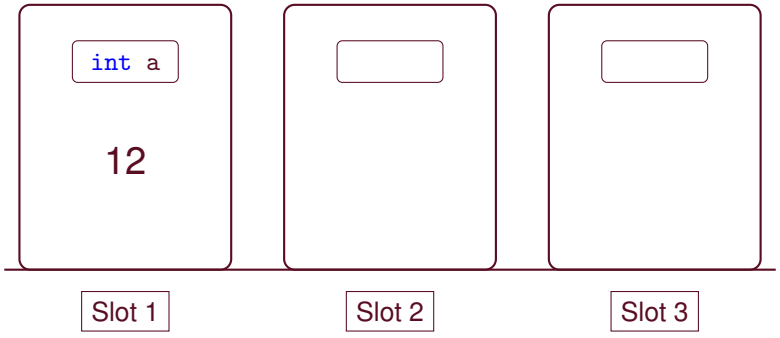

```
int a = 6;
```

Boxes in your memory



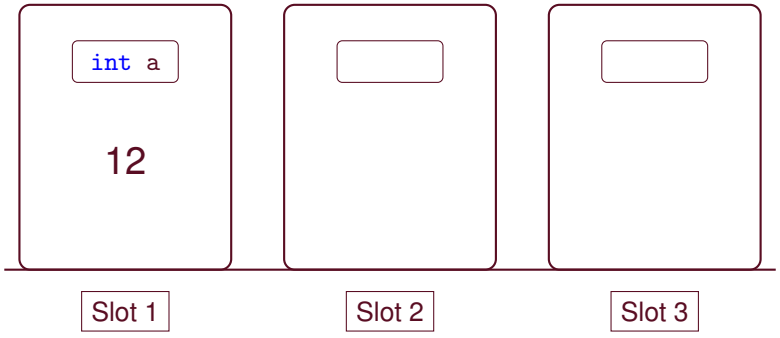

```
int a = 6;
```

Boxes in your memory




```
int a = 6;
a = 12;
```

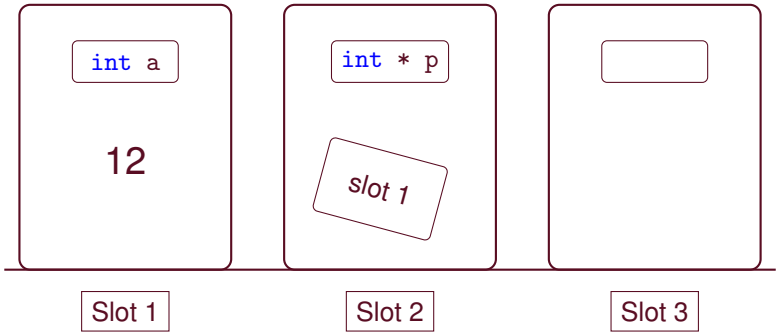
Boxes in your memory




```
int a = 6;
a = 12;
```

```
int * p = &a;
```

Boxes in your memory




```
int a = 6;
a = 12;
```

```
int * p = &a;
```

Pointers

The memory of a program is like a shelf with boxes positionned in slots; each box has a slot number, that is its **address.**

Pointers

The memory of a program is like a shelf with boxes positionned in slots; each box has a slot number, that is its **address.**

A **pointer** is a type of variable that stores the **address** of another variable.

Pointers

The memory of a program is like a shelf with boxes positionned in slots; each box has a slot number, that is its **address.**

A **pointer** is a type of variable that stores the **address** of another variable.

To declare a pointer, you write the type of the variable you'll store the address of, and add a $*$.

To get the address of a variable, you write a `&` before the variable's name.

Pointers

The memory of a program is like a shelf with boxes positionned in slots; each box has a slot number, that is its **address.**

A **pointer** is a type of variable that stores the **address** of another variable.

To declare a pointer, you write the type of the variable you'll store the address of, and add a `*`.

To get the address of a variable, you write a `&` before the variable's name.

```
int a = 12;
int * p = &a; // p contains the address of a
// we say that p points to a
```

Reaching the value pointed by a pointer

You can **dereference** a pointer to read the value contained in the variable it is pointing at.

Reaching the value pointed by a pointer

You can **dereference** a pointer to read the value contained in the variable it is pointing at.

```
int a = 12;
int * p = &a; // p contains the address of a
std::cout << "a = "  << *p;
```

Pointers initialization

When initializing a pointer, if you don’t assign it a value yourself, it will contain a random memory address. This means the pointer points at a memory spot on your computer that may contain anything. It could contain sensitive data.

Pointers initialization

When initializing a pointer, if you don't assign it a value yourself, it will contain a random memory address. This means the pointer points at a memory spot on your computer that may contain anything. It could contain sensitive data.
Rule of thumb: always initialize a pointer with either:

Pointers initialization

When initializing a pointer, if you don't assign it a value yourself, it will contain a random memory address. This means the pointer points at a memory spot on your computer that may contain anything. It could contain sensitive data.
Rule of thumb: always initialize a pointer with either:

- An address you have your hands on — i.e one of the program's variable address

Pointers initialization

When initializing a pointer, if you don't assign it a value yourself, it will contain a random memory address. This means the pointer points at a memory spot on your computer that may contain anything. It could contain sensitive data.
Rule of thumb: always initialize a pointer with either:

- An address you have your hands on — i.e one of the program's variable address
- The value zero.

Pointers initialization

When initializing a pointer, if you don't assign it a value yourself, it will contain a random memory address. This means the pointer points at a memory spot on your computer that may contain anything. It could contain sensitive data.
Rule of thumb: always initialize a pointer with either:

- An address you have your hands on — i.e one of the program's variable address
- The value zero.

Address zero or `nullptr`

The address 0 is a valid address; it points to *nothing*. Dereferencing address 0 will cause a crash, so always check `p != 0` before dereferencing a pointer.

Address zero or `nullptr`

The address 0 is a valid address; it points to *nothing*. Dereferencing address 0 will cause a crash, so always check `p != 0` before dereferencing a pointer.
You set a pointer to point to 0 with either one of the following:

```
int * p = 0;
int * p = NULL;
int * p = nullptr;
```

Copying a pointer 1/2

```
int a = 12;
int b = a;
a = 4;
// b still contains 12 because a and b are different boxes
```

Copying a pointer 1/2

```
int a = 12;
int b = a;
a = 4;
// b still contains 12 because a and b are different boxes
```

What happens if you copy a pointer?

Copying a pointer 1/2

```
int a = 12;
int b = a;
a = 4;
// b still contains 12 because a and b are different boxes
```

What happens if you copy a pointer?

```
int a = 12, b = 13;
int * p1 = &a;
int * p2 = p1;
*p1 = b;
std::cout << *p2 << std::endl;
```

Copying a pointer 1/2

```
int a = 12;
int b = a;
a = 4;
// b still contains 12 because a and b are different boxes
```

What happens if you copy a pointer?

```
int a = 12, b = 13;
int * p1 = &a;
int * p2 = p1;
*p1 = b;
std::cout << *p2 << std::endl;
```

Let's look here

Copying a pointer 2/2

What happens if you copy a pointer?

Copying a pointer 2/2

What happens if you copy a pointer?

```
int a = 12, b = 13;
int * p1 = &a;
int * p2 = p1;
p1 = &b;
std::cout << *p2 << std::endl;
```

Copying a pointer 2/2

What happens if you copy a pointer?

```
int a = 12, b = 13;
int * p1 = &a;
int * p2 = p1;
p1 = &b;
std::cout << *p2 << std::endl;
```

Let's look here

Practise time!

Data size

When you store media on your computer, it uses up computer storage. A file has a **size** in **bytes**.

Data size

When you store media on your computer, it uses up computer storage. A file has a **size** in **bytes**.
In the same way, a variable uses up program storage / memory. A variable also has a size, that depends on its *type*. This means the boxes on the shelf have *different sizes*.

Data size

When you store media on your computer, it uses up computer storage. A file has a **size** in **bytes**.
In the same way, a variable uses up program storage / memory. A variable also has a size, that depends on its *type*. This means the boxes on the shelf have *different sizes*.
You can get the size of a variable, that is the memory it uses in a program, with `sizeof`. It will give you the size in bytes.

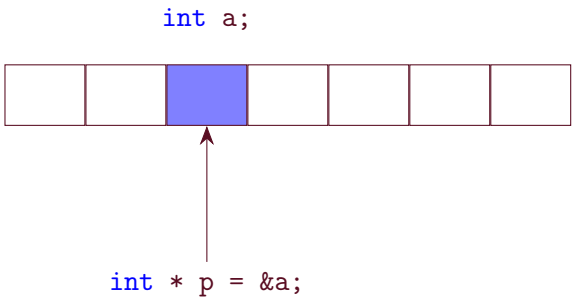
Incrementing a pointer 1/2

A pointer points to a spot in the program's memory. To increment a pointer is to make it point to the next spot in the program's memory.

```
int a;
```

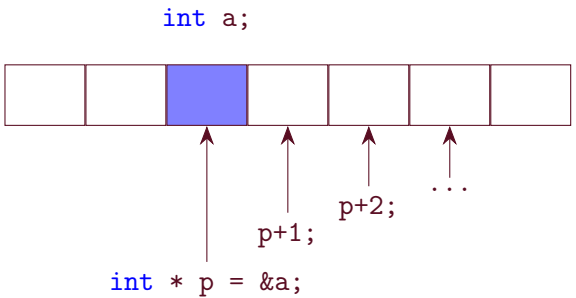
Incrementing a pointer 1/2

A pointer points to a spot in the program's memory. To increment a pointer is to make it point to the next spot in the program's memory.



Incrementing a pointer 1/2

A pointer points to a spot in the program's memory. To increment a pointer is to make it point to the next spot in the program's memory.



Incrementing a pointer 2/2

The drift of the pointer when it is incremented depends on the type of the spots it points to.

Incrementing a pointer 2/2

The drift of the pointer when it is incremented depends on the type of the spots it points to.
Look at this.

Why would we want to increment a pointer?

Remember `argv`?

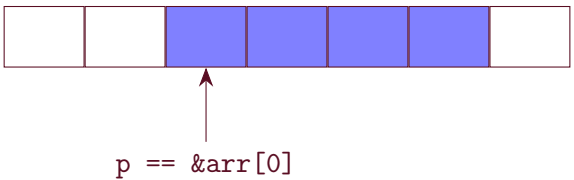
Pointers and arrays

If some elements are stored contiguously in memory, then a pair composed of a pointer and element count can represent an array.

Pointers and arrays

If some elements are stored contiguously in memory, then a pair composed of a pointer and element count can represent an array.

```
std::array<int, 4> arr {1,2,3,4};
int * p = arr.data();
size_t count = arr.size(); // == 4
```



Pointers and arrays

If some elements are stored contiguously in memory, then a pair composed of a pointer and element count can represent an array.

```
std::array<int, 4> arr
    {1,2,3,4};
int * p = arr.data();
for(int i = 0; i < arr.size();
     ++i){
    std::cout << arr[i]
        << std::endl;
}
```

```
std::array<int, 4> arr
    {1,2,3,4};
int * p = arr.data();
for(int i = 0; i < arr.size();
     ++i){
    std::cout << *(p+i)
        << std::endl;
}
```

C-style arrays

```
<type> arr [] = { ... };
```

C-style arrays are static arrays, very much like `std::array`. The difference is they don't have member functions like `size()`, so you have to keep track of their element count in a seperate variable.

C-style arrays

```
<type> arr [] = { ... };
```

C-style arrays are static arrays, very much like `std::array`. The difference is they don't have member functions like `size()`, so you have to keep track of their element count in a seperate variable.

C-style arrays and pointers are most often interchangeable once instanciated.

```
void foo(int * ptr);
int arr [] = {1,2,3,4};
foo(arr); // OK
```

Pointer to const

You may want to have a pointer to data but not allow the pointer to alter that data. In that case, you use a **pointer to const**.

Pointer to const

You may want to have a pointer to data but not allow the pointer to alter that data.

In that case, you use a **pointer to const**.

Like so: `<type> const *` or `const <type> *`.

Pointer to const

You may want to have a pointer to data but not allow the pointer to alter that data.

In that case, you use a **pointer to const**.

Like so: `<type> const *` or `const <type> *`.

The word `const` sits *beside the type*.

Pointer to const

You may want to have a pointer to data but not allow the pointer to alter that data.
In that case, you use a **pointer to const**.
Like so: `<type> const *` or `const <type> *`.
The word `const` sits *beside the type*.

```
int a = 10;
const int * p = &a;
*p = 8; // illegal, p is pointer to const
```

Const pointer

You may want a pointer to point a specific variable, and disallow changing the address it points to. In that case, you use a **const pointer**.

Const pointer

You may want a pointer to point a specific variable, and disallow changing the address it points to. In that case, you use a **const pointer**.

Like so: `<type> * const`.

Const pointer

You may want a pointer to point a specific variable, and disallow changing the address it points to. In that case, you use a **const pointer**.

Like so: `<type> * const`.

The word `const` sits *after the star*.

Const pointer

You may want a pointer to point a specific variable, and disallow changing the address it points to. In that case, you use a **const pointer**.

Like so: `<type> * const`.

The word `const` sits *after the star*.

```
int a = 10, b = 78;
int * const p = &a;
p = &b; // illegal, p is a const pointer
```

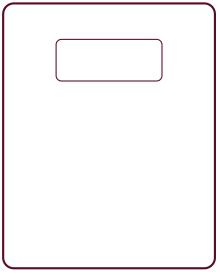
References vs pointers 1/3

References are only an *alias* to a variable. It's like putting a second label on a same box.

References vs pointers 1/3

References are only an *alias* to a variable. It's like putting a second label on a same box.

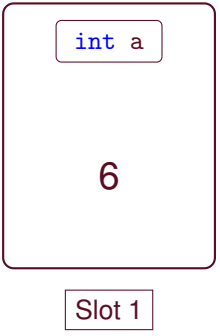
```
int a = 6;
```



References vs pointers 1/3

References are only an *alias* to a variable. It's like putting a second label on a same box.

```
int a = 6;
```

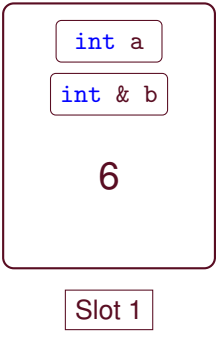


References vs pointers 1/3

References are only an *alias* to a variable. It's like putting a second label on a same box.

```
int a = 6;

int & b = a;
```



References vs pointers 2/3

References	Pointers
Have the same address as the variable they denote	Have their own address

References vs pointers 2/3

References	Pointers
Have the same address as the variable they denote	Have their own address
Cannot be empty, thus cannot exist beyond the scope of the variable they denote.	Will contain an address no matter what. If the object they point to is destroyed before them, the pointer becomes invalid but can still exist and be used.

References vs pointers 3/3

Because pointers can become invalid and references can't, the latter as safer to use than the former.

Rule of thumb: whenever possible, if you have the choice between the two, prefer references.

02 Algorithms on vectors and containers

Iterators

Containers such as `std::vector` and `std::array` provide **iterators**. They are very similar to pointers.

Iterators

Containers such as `std::vector` and `std::array` provide **iterators**. They are very similar to pointers.

```
std::vector<int> vec { 1, 2, 3, 4 };
int * pointer = &vec[0];
auto iterator = vec.begin();
```

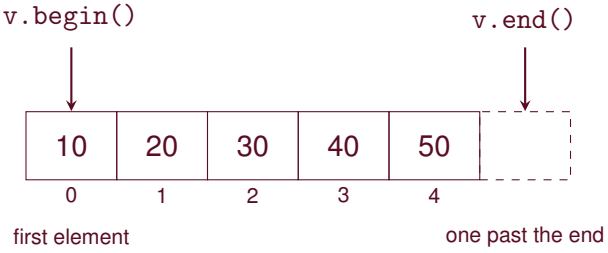
Iterators

Containers such as `std::vector` and `std::array` provide **iterators**. They are very similar to pointers.

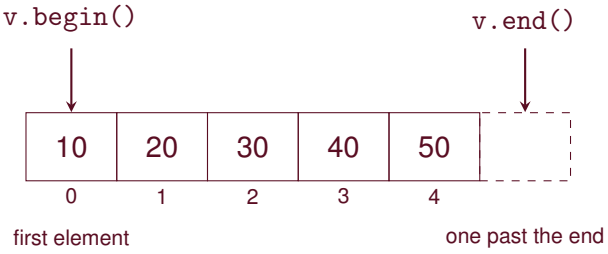
```
std::vector<int> vec { 1, 2, 3, 4 };
int * pointer = &vec[0];
auto iterator = vec.begin();

// prints first element of vec
std::cout << *ptr << std::endl;
std::cout << *iterator << std::endl;
 // prints second element
std::cout << *(ptr + 1) << std::endl;
std::cout << *(iterator + 1) << std::endl;
```

Begin and end



Begin and end



Traversing a `std::vector` using iterators:

```
for(auto iter = v.begin(); iter != v.end(); ++iter){
    // ...
}
```

STL's algorithms 1/2

The C++ *Standard Template Library* (STL) gathers a collection of functions to facilitate the use of containers such as `std::vector`.

STL's algorithms 1/2

The C++ *Standard Template Library* (STL) gathers a collection of functions to facilitate the use of containers such as `std::vector`.
Some examples:

```
// sorts in non-descending order
std::sort(v.begin(), v.end());
// copy elements from [v_src.begin(), v_src.end()-1] at the back of
    v_dst
std::copy(v_src.begin(), v_src.end(), std::back_inserter(v_dst));
```

STL's algorithms 1/2

The C++ *Standard Template Library* (STL) gathers a collection of functions to facilitate the use of containers such as `std::vector`.
Some examples:

```
// sorts in non-descending order
std::sort(v.begin(), v.end());
// copy elements from [v_src.begin(), v_src.end()-1] at the back of
    v_dst
std::copy(v_src.begin(), v_src.end(), std::back_inserter(v_dst));

// search for the first occurence of 4 if any in v. If v contains a
    4, found will point at it; otherwise, found points at v.end().
auto found = std::find(v.begin(), v.end(), 4);
```

STL's algorithms 2/2

There are plenty of algorithms already implemented in the STL. If you need to do some basic operation on a container, always try to find if it hasn't been implemented in the STL. They are referenced here: `https://en.cppreference.com/cpp/header/algorithm`.

Practise time!

03 Accessing text files

Reading from text files: an example

Let’s have a look at an example.

Reading from text files

`std::ifstream` is an **input file stream**. You can use it to **read** from a file.

Reading from text files

`std::ifstream` is an **input file stream**. You can use it to **read** from a file.

You can either read a file line by line with `std::getline`.
Or you can use the operator >> to read data into variables. You have to be careful about type (mis-)match.

Writing to text files: an example

Let’s have a look at an example.

Writing to text files

`std::ofstream` is an **output file stream**. You can use it to **read** from a file.

Writing to text files

`std::ofstream` is an **output file stream**. You can use it to **read** from a file.

You can write anything to a file using << just like you'd write to the terminal output with `std::cout << ...`.

Practise time!

04 Reading and modifying audio files

What are sound files? Briefly

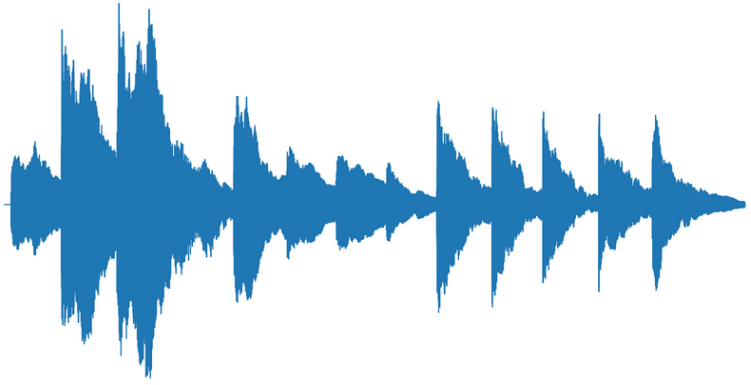


Figure: Audio waveform

Audio files and how to manipulate them

Sound can be represented as a collection of `float` values all in the range $[-1, 1]$.

Audio files and how to manipulate them

Sound can be represented as a collection of `float` values all in the range $[-1, 1]$.
We can use the libsnfile library to open, read from and write to audio files.

Practise time!

See you next time!