Cours de C++

Gestion de la mémoire et structure bas-niveau

Cécile Braunstein
cecile.braunstein@lip6.fr

Low-level data structures

Until now we used :

- Variables
- Reference
- Containers

How it works ?

Use low-level techniques and tools.

- Underlie the standard library
- Close to the way hardware works

Pointers

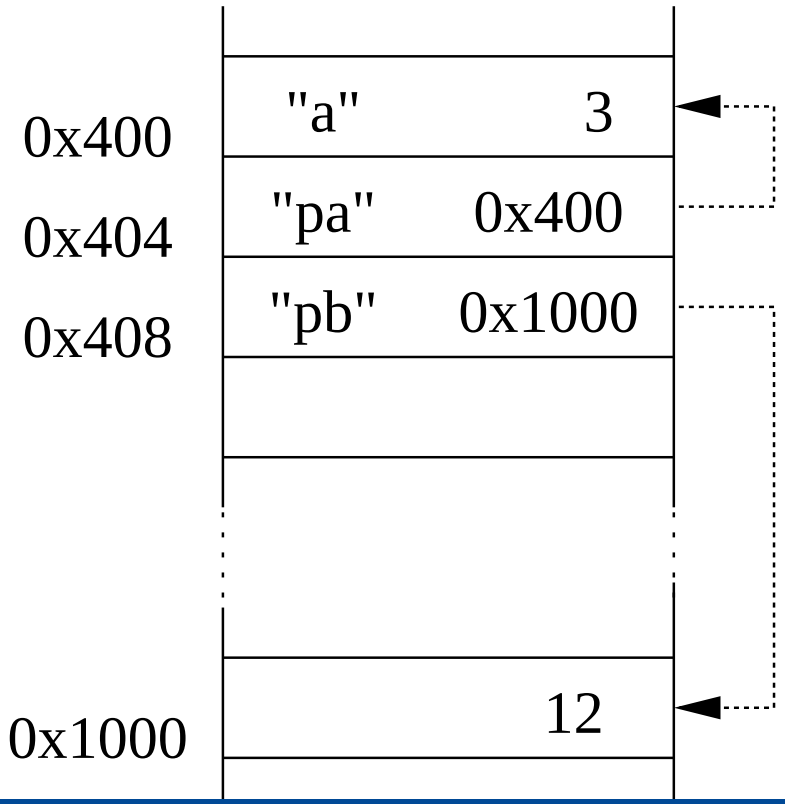
Definition

A pointer is a value that represents the address of an object.

Every distinct object has a unique address. It's the the part of the computer's memory that contains the object.

```
int main()
{
  int a = 3;
  int *pa ;
  int *pb ;

  pa = &a;
  pb = (int*)malloc(sizeof(
      int));
  *pb = 12
  return 0;
}
```



Operators on pointers

`&x`	: address operator
`*px`	: dereference operator
`T* p`	: declaration of a pointer to `T` (`*p` has a type `T`)
`NULL`	: constant value, differs from every pointer to any object

Operations on pointers

Exercice

What is the output of this program. We assume that `&x = 0xbf84e7b8`

```
#include<iostream>
using namespace std;
int main(){
  int x = 5;
  int* p = &x;
  cout << "x = " << x << endl;
  cout << "p = " << p << " ; *p = " << *p << endl;

  *p=6;
  p = p + 1;
  cout <<"x = " << x << endl;
  cout << "p = " << p << " ; *p = " << *p << endl;
  return 0;
}
```

References vs. Pointers

```
#include<iostream>
using namespace std;
void increment(int& v)
{ v++;}
int main(){
 int a = 3 ; int* pa;
 int & ra = a;
 pa = &a ; ra = 4;
 increment(a);

 cout << "a= " << a
 <<" &a= " << &a<< endl;
 cout << "*pa= " << *pa
 << " pa= " << pa << endl;
 cout << "ra= " << ra
 <<" &ra= "<< &ra << endl;
 return 0;
}
```

Reference's properties

- A reference is a pointer self-dereferenced
- The assignation to a variable is only done at the declaration and can never be changed.
- Most use for the parameters list of a function

The operator & :
in a type declaration = a reference
in an expression = an address

Pointers to functions

What we cannot do with a function

A function is not an object :

- No copy
- No assignment to a value
- No passing as argument directly

A program cannot :

- Create a function
- Modify a function

Only compilers can do that

What we can do with a function

- Call the function
- Take its address

Pointers to functions

Using function as argument to another function

Declaration

```
int (*fp)(int);
```

`fp` is a pointer to function that takes an `int` argument and returns an `int` result.

Use

- All use that is not a call is assumed to be taking its address, even without &.
- We can call a pointer to a function without dereferencing it.

Pointers to function

Example

Example

```
int increment(int n) {return n = n + 1;}
```

fp = &increment ⇔ fp = increment

i = (*fp)(i) ⇔ i = fp(i)

Example

```
template <class InIte, class Pred>
InIte find_if(InIte begin, InIte end, Pred f)
{
 while(begin != end && !f(*begin))
  ++begin;
 return begin;
}
```

Array

```
#include<iostream>
using namespace std;
int main ()
{
  int ta[3] = { 1, 2, 3 };
  int *pta;

  cout << "ta[1] := " << ta[1] << endl;
  cout << "ta[1] := " << *(ta+1) << endl;

  pta = ta;
  cout << "pta[1] := " << pta[1] << endl;
  cout << "pta[1] := " << *(pta+1) << endl;

  return 0;
}
```

Array initialization

```
const int DIM = 3;
double tab[DIM] = {1,2,3};

double number[] = {
1,2,3,4,5,6}
```

Number of elements

The size of an array have to be known at compile time.
For implicit initialization :

```
size_t n= sizeof(number)/sizeof(*number);
```

Automatic memory management

Local variables

The program allocates memory when it encounters the definition of the variable
The program deallocates that memory at the end of the block containing the definition.

```
int* invalid_pointer()
{
  int x;
  return &x; // never !
}
```

Automatic memory management

Local variables

The program allocates memory when it encounters the definition of the variable
The program deallocates that memory at the end of the block containing the definition.

```
int* static_pointer()
{
  static int x;
  return &x; // never !
}
```

Static allocation

`static`

- A variable declares as `static` will be allocate once and only once
- The allocation is performed before the function is called (or the object is instantiated)
- It will deallocate at the end of the run.

Static member function

- No need to be applied to an object
- Can have access only to static members

Dynamic allocation

Allocate

```
int* p = new int(42);
```

- Allocate a new object of type `int`
- Initialize the object to 42
- Cause `p` to point to that object

The object stays around until it is deleted or the program ends.

Deallocate

```
delete p;
```

- Frees space memory used by `*p`
- Invalids `p`

Deleting a zero pointer has no effect.

Allocating and deallocating an array

```
T* p = new T[n]
delete[] p
```

Allocation

- Allocates and default-initializes an array
- Returns a pointer to the first element in the array

Deallocation

- Destroys the objects in the array
- Frees the memory used to hold the array
- Invalids the pointer p ;

Life of a class

1 – Allocate memory
2 – Initialization (constructor) $\Longleftrightarrow$ Call **new**
3 – Use the object
4 – Clean the memory (destructor)
5 – Give the memory back $\Longleftrightarrow$ Call **delete**

Life of a class

1 – Allocate memory
2 – Initialization (constructor) $\Longleftrightarrow$ Call **new**

3 – Use the object

4 – Clean the memory (destructor)
5 – Give the memory back $\Longleftrightarrow$ Call **delete**

Life of a class

1 – Allocate memory
2 – Initialization (constructor) $\Longleftrightarrow$ Call **new**

3 – Use the object

4 – Clean the memory (destructor)
5 – Give the memory back $\Longleftrightarrow$ Call **delete**

Life of a class

1 – Allocate memory
2 – Initialization (constructor) $\Longleftrightarrow$ Call **new**

3 – Use the object

4 – Clean the memory (destructor)
5 – Give the memory back $\Longleftrightarrow$ Call **delete**

Life of a class

1 – Allocate memory
2 – Initialization (constructor) $\Longleftrightarrow$ Call **new**

3 – Use the object

4 – Clean the memory (destructor)
5 – Give the memory back $\Longleftrightarrow$ Call **delete**

Life of a class

1 – Allocate memory
2 – Initialization (constructor) $\Longleftrightarrow$ Call **new**

3 – Use the object

4 – Clean the memory (destructor)
5 – Give the memory back $\Longleftrightarrow$ Call **delete**

Life of a class

1 – Allocate memory
2 – Initialization (constructor) $\Longleftrightarrow$ Call **new**

3 – Use the object

4 – Clean the memory (destructor)
5 – Give the memory back $\Longleftrightarrow$ Call **delete**