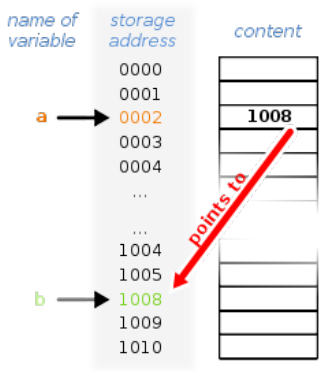
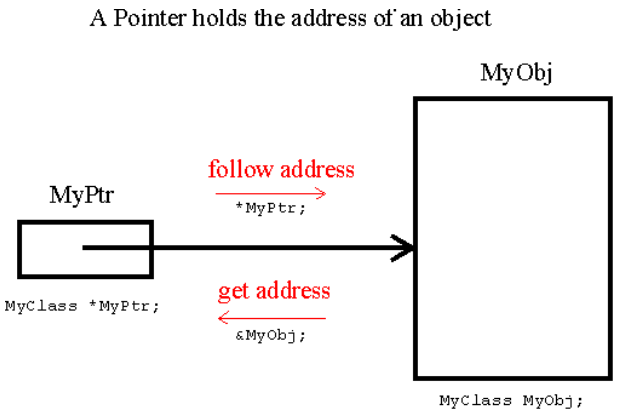
C++ lab 2

Recap of what you should have learned during the second lab:

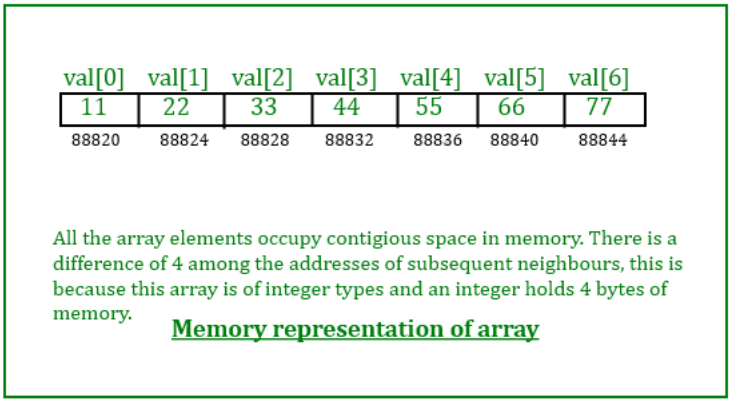
1. Dynamic memory allocations

- Use the `new` operator to allocate dynamically a unique object of a given type. With this operator you can also directly call any of the constructor of the object:
  - `int* val = new int;`
  - `PPoint* point = new PPoint(1,5);`





- Use the `new[]` operator to allocate dynamically an array of objects of a given type. With this operator you cannot call the constructor of the objects, the type have to be default-constructible:
  - `int* values = new int[size];`
  - `PPoint* points = new PPoints[size];`



- For every call to `new` you should have a matching call to `delete` to free allocated memory.
- Similarly, for every call to `new[]` you should have a matching call to `delete[]`.
- If `new` and `delete` operator do not match, it will result in an undefined behaviour.
- Do not forget to implement a destructor that frees memory for each class that dynamically allocates memory in its constructors.
- Copying a pointer will not copy the pointed object(s) ! To copy the pointed object(s) you need to dynamically allocate a new array and use functions like `memcpy` or `strcpy` to copy the array contents.
- This is why you have to pay attention to define a custom copy-constructor (and as we will see later, a custom `operator=`). The default copy-constructor will just copy the pointers.
- Freeing two times the same pointer will lead to a `segfault` and crash your program.
- You can check all memory leaks in your program by using the `valgrind` tool:
  - `valgrind --leak-check=full [program] [arguments]`

2. Pass by value vs pass by reference

- When you pass an argument by value, is is copied using the `copy-constructor` prior to the function call:

```
int duration(Date d0, Date d1) { /* implementation */ };
Date date0, date1;
duration(date0, date1);
```

is equivalent to the inlined function:

```
Date date0, date1;
{
    Date d0(date0); // copy-constructor of Date
    Date d1(date1); // (the whole object is copied)
    /* implementation */
}
```

- When you pass an address by using a pointer, you pass the pointer by value:

```
int duration(Date* d0, Date* d1) { /* implementation */ };
Date date0, date1;
Date* date0_ptr = &date0;
Date* date1_ptr = &date1;
duration(date0_ptr, date1_ptr);
```

is equivalent to the inlined function:

```
Date date0, date1;
Date* date0_ptr = &date0;
Date* date1_ptr = &date1;
{
    Date* d0(date0_ptr); // copy-constructor of Date*
    Date* d1(date1_ptr); // (only the memory address is copied)
    /* implementation */
}
```

- As opposed to pure `C`, `C++` offers pass-by-reference capabilities:

```
int duration(Date& d0, Date& d1) { /* implementation */ };
Date date0, date1;
duration(date0, date1);
```

is equivalent to the inlined function:

```
Date date0, date1;
{
    Date& d0(date0); // constructor Date&(Date)
    Date& d1(date1); // (no copy)
    /* implementation */
}
```

This acts as if you passed the arguments by address but without the pointer notations.

- The type of the reference is the name of the type followed by `&`.
- For raw `C` arrays you can only use pointers.

3. Const correctness
   - `const correctness` means using the keyword `const` to prevent constant objects from getting mutated.
   - Example of `pass-by-pointer-to-const`:

```
int duration(const Date* d0, const Date* d1) { /* implementation */ };
```

   - Example of `pass-by-reference-to-const`:

```
int duration(const Date& d0, const Date& d1) { /* implementation */ };
```

   - You can always cast a `non-const` object to a `const` object whenever you want.
   - You cannot convert directly a `const-reference` to a `reference`.
   - You cannot convert directly a `pointer-to-const` to a `pointer-to-non-const`.
   - It is possible with the `const_cast` operator but modifying a `const` object through a `non-const` access leads to undefined behaviour.
   - How to declare that a member function will not modify the current object ?

```
class Date{
    private:
        int day, month, year;
    public:
        int get_day() {
            // Here this is of type 'Date*'
            return this->day;
        };
};
```

The class method `get_day` is equivalent to the external function:

```
int get_day(Date* this);
```

and we would like to declare that `this` is a pointer to a constant `Date` object:

```
int get_day(const Date* this);
```

This can be done by adding the `const` keyword after the method signature:

```
class Date{
    private:
        int day, month, year;
    public:
        int get_day() const {
            // Here this is of type 'const Date*'
            return this->day;
        };
};
```

4. Documenting your code
   - You can use a documentation generator like `doxygen` to generate a documentation for your code.
   - Code that you wrote 6 months ago is often indistinguishable from code that someone else has written.
   - See the `exercice1` for an overview of `doxygen` capabilities (use `make html`).
5. Miscellaneous
   - The C header `<cstring>` defines `strcpy` and `memcpy`.
   - You can get the size of a type or object in bytes by using the `sizeof` operator.
   - To generate random integer values you can use:
     - `std::rand()` defined in the `C` header `<cstdlib>`.
     - Do not forget to use `std::srand()` to initialize the random seed in your main function.
   - To check if a floating point number is a `NaN` you can use:
     - `val==val` will return False if and only if val is a `NaN`.
     - `std::isnan(val)` defined in the `C` header `<cmath>`.
   - Add the `-g` compiler option to your `CXXFLAGS` variable to debug your program. It will give you the exact file and line causing the problem in `valgrind` and `gdb`.