# How to define methods?

We can define methods in two places.

1. Methods defined inside the class declaration (we did that last time)

```
class Person
{
    string m_name
    int m_age;
public:
    void remember (string name, int age)
    {
        m_name = name;
    }

    void print()
    {
        cout << m_name << " , years old: „ << m_age << endl;
    }
};
```

# How to define methods?

2. Methods defined outside a class (in a compilation unit – a cpp file)

Person.h:

```
class Person
{
    string m_name
    int m_age;
 public:
    void remember (string name, int age)
    void print ();
};
```

Person.cpp:

```
void Person::remember (string name, int age)
{
    m_name = name;
}
void Person::print()
{
    cout << m_name << " , years old: „ << m_age << endl;
}
```

# Methods defined inside versus outside?

Each method defined inside a class is automatically *inline*.

Each method defined outside a class is not automatically *inline*.

Methods defined as *inline*:

“The original intent of the inline keyword was to serve as an indicator to the optimizer that inline substitution of a function is preferred over function call, that is, instead of executing the function call CPU instruction to transfer control to the function body, a copy of the function body is executed without generating the call. This avoids overhead created by the function call (copying the arguments and retrieving the result) but it may result in a larger executable as the code for the function has to be repeated multiple times. “ – cppreference.com

**How to declare methods:**

If a method definition has only few lines then we can define it inside a class but in other cases we should define methods outside to keep definition of a class clean.

# What is the *this* pointer?

A method has access to fields but how each instance of a given class is distinguished?

```
void Person::remember (string name, int age)
{
    m_name = (name != "" ? name : "Anonim");
    m_age = age;
}
```

Imagine a situation when we have in memory few instances of the class *Person:* student1, student2, professor, pilot. We have four strings for names and four variables of type int to store information about age. Inside the definition of the method *remember ()* there is written only *m_age* and *m_name*. The method is in the memory only once. How then it knows to which address of memory refers to while operating on a different instance? The answer is simple, in behind the address of an instance is send and stored in a point called *this*.

# What is the *this* pointer?

In fact the definition of the method looks like:

```
void Person::remember (const char * name, int age)
{
    this->m_name = (name != "" ? name : "Anonim");
    this->m_age = age;
}
```

We could always writer *this* but our compilator is doing this for us automatically. Note that, the *this* pointer is of type: **`Type const *`**

# How to pass an object as a function parameter?

Normally, each object is passed to a function by value in a similar way as for types *int* or *float*.

*Corollary:*

Passing many huge object as parameters of functions can take time and slow down our programs.

C style solution:

```
void present(const Person * somebody)
{
    cout << "I would like to introduce you \n"
    somebody->print();
}
```

# Passing by reference

If we pass an object by a reference then we ensure that it is not copied. A given function has access to the original object via a different name.

```
void present(const Person & somebody)
{
    cout << "I would like to introduce you \n"
    somebody.print();
}
```

Right now, the function does not copy an object but creates a new name of it with a scope bounded by the scope of the method present.

# Constructor: Introduction

Let's start with the following class:

```
class Number
{
        int m_number;

    public:

        void store(int number)
        {
            m_number = number;
        }

        int get()
        {
            return m_number;
        }
};
```

# Constructor: Introduction

If we would like to store a number in a variable then we will write:

```
int number = 5;
```

In the case of the class *Number* we have to write:

```
Number number;
number.store ( 5 );
```

We see that using our class is not as comfortable as using build-in types. We can overcome this by using constructors.

# Constructor: Introduction

Here a modified version of the class:

```
class Number
{
        int m_number;

    public:
        Number(int number)
        {
            m_number = number;
        }
        void store(int number)
        {
            m_number = number;
        }

        int return()
        {
            return m_number;
        }
};
```

How to use constructors:

```
Number number = Number(5);
```

or

```
Numer number(5);
```

# Name overloading: Introduction

In C++ we can overload functions. This means that we can have several functions of the same name. Nevertheless, they have to be different by their signatures – list of parameters. During the compilation time, our compiler checks lists of parameters and their types and try to match a function to be called.

# Name overloading: Introduction

```
class Clock
{
    int m_sec;
public:
    Clock (int sec) {m_sec = sec;}
    Clock(int min, int sec){m_sec = 60 * min + sec;}
    Clock (int ho, int min, int sec)
    {
        sec = 3600 * ho + 60 * min + sec;
    }
};

int mian()

{

    Clock time1(5);

    Clock time2(5,3);

    Clock time3(1, 2, 30);

}
```

# Initialization list

```
ClassName::ClassName(args) : initalization_list
{
// constructor body
}

class ABC
{
   const int y;
   float x ;
   char c ;

// constructor declaration
   ABC(float pp, int dd, char cc);
};

// constructor definition
ABC::ABC(float pp, int dd, char cc) : y(dd), c(cc)
{
   x = pp;
}
```

# Initialization list

Each constructor call has two stages:

Stage 1: initialization via initialization list

Stage 2: constructor block

We can initialize non-const fields in two ways:

- initialization list,
- constructor body,

Nevertheless, fields of type `const` can be initialized only via initialization list.

# Destructor: Introduction

A destructor is a method called when an object is going to be removed from memory.

Each class can have only one destructor since it has no parameters and cannot be overloaded.

Names of destructors are of format: **`~class_name()`**.

# Static fields

A static field is a field which is placed in memory only once independent to number of instances of a given class. It is a field which stores information about a class, common for all the instances. A static field is defined when a program is executed, and exists before the first instance of a class is created.

A static field is better than a variable of the global or file scope – a so-called global variable:

- A global scope variable is accessible without any control
- Many global scope variables can cause name collisions

**Example**:

```
class A
{
 public:
    int x;
    static int field;
};
```

# Static fields

The declaration of a static field inside a class is not its definition. The definition has to be placed outside a class – similarly to global scope variables which have to be placed outside functions.

```
int A::field = 0;
```

A static field can be **`private`** and then after its initialization it can be accessed only from inside the class.

Ways to access a static field:

**`A::field`**

**`object.field`**

**`pointer->field`** // where pointer is of the type: **`A * pointer`**;

# Static methods

A static method can access only static fields and we can call statics methods without creating objects.

What do we gain by having a static function in a class? If we have a private static field and we would like to change it before creating the first instance then we can do it only via a static method.

# Properties of static methods

A static method is related to a class and not to any object.

Ways to call:

```
object.method();
```

```
A::Method();
```

# Examples of use of static methods

- An inter-object communication. By change of a static field an object can inform others, of the same class, that a given event occurred.
- All objects have the same state which changes for all of them
- An object counter
- Resources sharing, for example, all objects use the same file

# Methods of the type const

Methods of type const cannot modify non-const fields.

A constant object:

```
const Bicycle example; // similar to eg, const float pi = 3.14
```

Fields of a constant object can be initialized by a constructor but after initialization they cannot be modified – this is not true but we will not go into details.

Note that for a constant object we can call only const methods – methods which do not change fields and are declared as const.

```
string Person::getName () const
{
    return m_name;
}
```

# Methods of the type volatile

The other qualifier in C and C++, volatile, indicates that an object may be changed by something external to the program at any time and so must be re-read from memory every time it is accessed. When a function is of the type volatile then this means that such a function will not copy data to the cache memory and it will always work on the original memory.

When an object is defined as of the type volatile then we can only call for it these methods which are declared as volatile.

```
volatile ChemicalReaction chainReaction;

void B::example () volatile
{
    //...
}
```

Note that, code which use `volatile` cannot be optimized by use of the cache memory.

**Question.** Can we define a method which is both `const` nad `volatile`? Think, test and send a mail with your answer.