C++
Classes

Goals

- Learn about references
- Learn how to define classes in C++
- Understand the difference between struct and class
- Understand constructors
- Understand how the friend keyword works

Quick reading
- Read and try to grasp the main ideas

Read
- Read and understand the explained concepts

Study
- Read, understand and remember the concepts, the rules and the principles.

Don't be afraid to try (compile, execute, modify, debug) the proposed examples!

Why on the stack

- As a programmer, allocating on the stack is trivial: no explicit allocation and deallocation
  - Objects on the stack are freed when we exit their scope
    - Local scope: end of the function
    - Class scope: when the instance of the class is destroyed
    - Namespace scope: when the program ends

References

- In C++ we can create alias to variables that we call **references**

```
int& c{a}; // Reference to an int (alias for a)
```

- In contrast to a pointer the reference cannot be modified to point to another variable
  - It's useful to avoid copies when passing parameters (and avoid pointers)

Example

```
#include <iostream>

using namespace std;

int main(void)
{
	int a;
	int b{13};
	int& c{a};

	a = 4;
	cout << "a=" << a << " b="  << b << " c="  << c << endl;
	a = b;
	cout << "a=" << a << " b=" << b << " c="  << c << endl;

	c = 7;
	cout << "a="  << a << " b=" << b << " c="  << c << endl;
}
```

Pass by value, pass by reference

```
int multiply_byvalue(int a, int b)
{
        return a*b;
}

int multiply_byref(int& a, int& b)
{
        return a*b;
}


int main(void) {
        int x{4}, y{3};
        multiply_byvalue(x,y);
        multiply_byref(x,y);
}
```

a and b are copies of the values passed to the function

a and b are references to the variables passed to the function

Manipulating parameters

```
#include <iostream>
using namespace std;

void cswap(int* x, int* y) {
        int temp{*x};
        *x = *y;
        *y = temp;
}
void swap(int& x, int& y) {
        int temp{x};
        x = y;
        y = temp;
}
int main(void) {
        int a{13}, b{17};
        cout << "a=" << a << " b=" << b << endl;
        cswap(&a, &b);
        cout << "a=" << a << " b=" << b << endl;
        swap(a,b);
        cout << "a=" << a << " b=" << b << endl;
}
```

Using pointers

Using references

Modifying an array

```
#include <iostream>

using namespace std;

int main(void) {
        int myarray[5] { 1, 5,
                         3, 6, 2};

        for (auto i=0; i<5; i++) {
                myarray[i] += 1;
        }

        for (auto& i : myarray) {
                cout << i << endl;
        }
}
```

Solution 1

```
#include <iostream>

using namespace std;

int main(void) {
        int myarray[] { 1, 5,
                         3, 6, 2};

        for (auto& i : myarray) {
                i++;
        }

        for (auto& i : myarray) {
                cout << i << endl;
        }
}
```

Solution 2

Returning references

```
#include <iostream>

using namespace std;

int& right(int& x, int& y)
{
	return x > y ? x : y;
}

int& wrong(int& x, int& y)
{
	int temp;
	if (x > y)
		temp = x;
	else
		temp = y;
	return temp; // Error! Temp is a local variable!
}
int main(void)
{
	int a{13}, b{17};
	cout << "a=" << a << " b=" << b << " max=" << right(a,b) << endl;
	cout << "a=" << a << " b=" << b << " max=" << wrong(a,b) << endl;
}
```

Lvalue, Rvalue

- Each experession in C++ can be either an
  - **Lvalue**
    - Object with a name (for example, variables), a precise address in memory (i.e. i can use **&** to get it)

    Can be passed by value or as reference
  - **Rvalue**
    - Temporary value that does not exist after the expression has finished using it

    Can be passed by value or as const reference or r-value reference (&&) - Requires C++ 11

```
int x = 7 + 10;
```

Lvalue **Rvalue**

Pass by reference of an Rvalue

```
#include <iostream>

using namespace std;

int f(int& x)
{
	return x;
}

int main(void)
{
	int a{13};
	f(a);
	f(6); // Error, 6 is an rvalue
	f(f(a)); // Error, f(a) is an rvalue
}
```

Reference to a const Rvalue

```
#include <iostream>

using namespace std;

int f(const int& x)
{
        return x;
}

int main(void)
{
        int a{13};
        f(a);
        f(6);
        f(f(a));
}
```

Passing an Rvalue reference

```
#include <iostream>

using namespace std;

int f(int& x)
{
	return x;
}

int f(int&& x)
{
	return x;
}

int main(void)
{
	int a{13};
	f(a);
	f(6); // calls the second version
	f(f(a)); // calls the second version
}
```

Java Reference vs C++ Reference

```java
class Esempio0 {

	public static void fun(int x) {
		x = 1;
	}

	public static void main(String[] args) {
		int s = 0;
		fun(s);
		System.out.println(s);
	}

}
```

```cpp
#include <iostream>
#include <string>

using namespace std;

void fun(int x) {
	x = 1;
}

int main() {
	int s = 0;
	fun(s);
	cout << s << endl;
}
```

Java Reference vs C++ Reference

“In Java simple values are passed by value"

Java Reference vs C++ Reference

“… and other types are passed by... ?”

Java Reference vs C++ Reference

```java
import java.lang.String;

class Example {

	public static void fun(String x) {
		x = "hello world";
	}

	public static void main(String[] args) {
		String s = "hello moon";
		fun(s);
		System.out.println(s);
	}

}
```

```cpp
#include <iostream>
#include <string>

using namespace std;
// Note: we ignore the memory leak

void fun(string* x) {
	x = new string{"hello world"};
}

int main() {
	string* s = new string{"hello moon"};
	fun(s);
	cout << *s << endl;
}
```

Java Reference vs C++ Reference

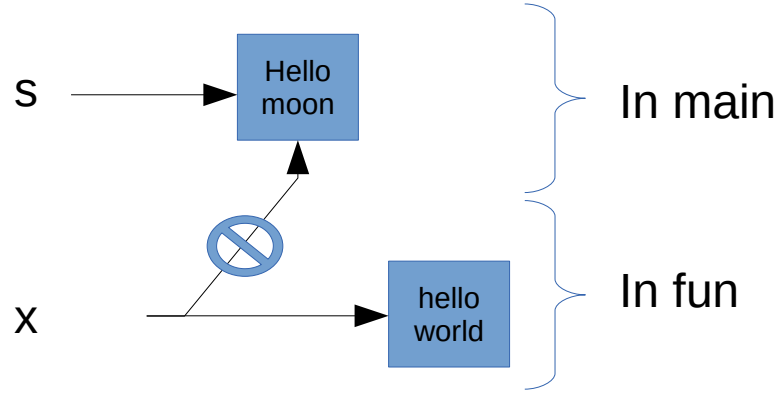
```java
import java.lang.String;

class Esempio {

	public static void fun(String x) {
		x = "hello world";
	}

	public static void main(String[] args) {
		String s = "hello moon";
		fun(s);
		System.out.println(s);
	}

}
```

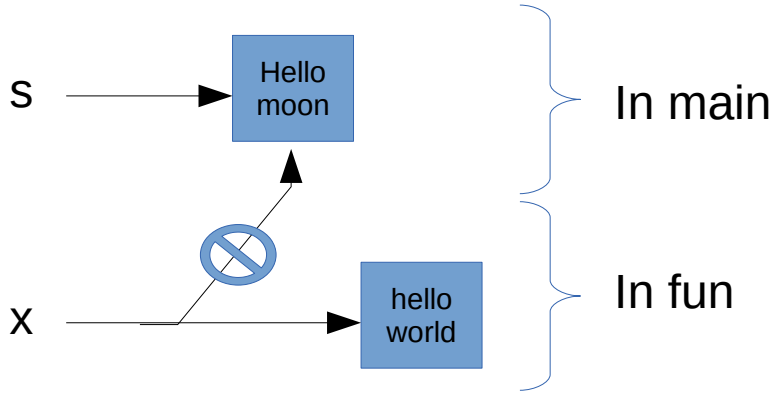
```cpp
#include <iostream>
#include <string>

using namespace std;

void fun(string& x) {
	x = "hello world";
}

int main() {
	string s = "hello moon";
	fun(s);
	cout << s << endl;
}
```

Java Reference vs C++ Reference

```java
import java.lang.String;

class Esempio {

	public static void fun(String x) {
		x = "hello world";
	}

	public static void main(String[] args) {
		String s = "hello moon";
		fun(s);
		System.out.println(s);
	}

}
```

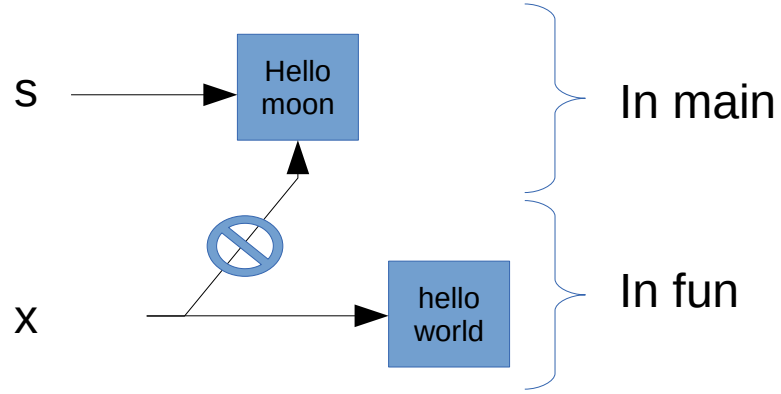
```cpp
#include <iostream>
#include <string>

using namespace std;
// Nota: Ignoriamo i memory-leak

void fun(string* x) {
	x = new string{"hello world"};
}

int main() {
	string* s = new string{"hello moon"};
	fun(s);
	cout << *s << endl;
}
```



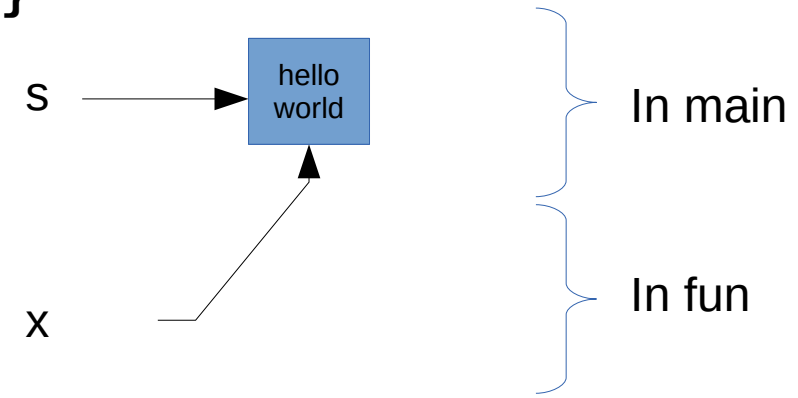


Java Reference vs C++ Reference

```java
import java.lang.String;

class Esempio {

	public static void fun(String x) {
		x = "hello world";
	}

	public static void main(String[] args) {
		String s = "hello moon";
		fun(s);
		System.out.println(s);
	}

}
```




```cpp
#include <iostream>
#include <string>

using namespace std;

void fun(string& x) {
	x = "hello world";
}

int main() {
	string s = "hello moon";
	fun(s);
	cout << s << endl;
}
```



Java Reference vs C++ Reference

Some people will say incorrectly that objects are passed "by reference." In programming language design, the term pass by reference properly means that when an argument is passed to a function, the invoked function gets a reference to the original value, not a copy of its value. If the function modifies its parameter, the value in the calling code will be changed because the argument and parameter use the same slot in memory. **The Java programming language does not pass objects by reference; it passes object references by value.**

J. Gosling

Struct, enum

```
enum Sex {
        Man, Woman
};
```

typedef not necessary

```
enum class Sex {
        Man, Woman
};
```

Not automatically converted to int (strongly typed)

```
struct Person {
        string first; string last; Sex s; unsigned int age;
};
```

typedef not necessary

struct

```
#include <string>

struct Fraction {
    int numerator, denominatore;
};
```

```
#include <iostream>
#include "fraction.h"

using namespace std;

void print(const Fraction& f) {
    cout << f.numerator << "/" <<
f.denominator << endl;
}

int main()
{
    Fraction f1;
    f1.numerator = 2;
    f1.denominator = 3;
    print(f1);
    f1.numerator = 7;
    f1.denominator = 3;
    print(f1);
}
```

What's missing?
Encapsulation/data-hiding

Combine data and methods, hide implementation details

class

- It's similar to a structure, but it's defined with the keyword **`class`**
- Like with structures, for a class *T* we typically:
  - Put the definition in the header file *T.h*
  - Write the implementation goes into *T.cpp*

Defining a class

```
class Fraction {
    public:
        int num() const;
        void num(int numerator);
        int den() const;
        void den(int denominator);

    private:
        int m_numerator {0}, m_denominator {1};
};
```

Access level

Declares that the method does not change the state of the object

Access level

Initialization

Access levels

- **Public** – Fields and methods accessible from everyone that nows the structure of the class (i.e has included the correct header file)
- **Protected** – Accessible to the member of the class and all derived classes
- **Private** – accessbile only with the class itself
  - Note: two objects of the same class can access their private fields/methods

Struct vs Class

- In C++ you can use either **struct** or **class**
  - The only difference lies in the default access level
    - **class**: is private by default
    - **struct**: is public by default

```
struct T { ... }
```

Is equivalent to

```
class T { public: ... }
```

Inline implementation

```
class Fraction {
    public:
        int num() const { return m_numerator; }
        void num(int numerator) {
                  m_numerator = numerator; };
        int den() const { return m_denominator; }
        void den(int denominator) {
                  m_denominator = denominator; };

    private:
         int m_numerator {0}, m_denominator {1};
};
```

Methods are implemented inside the class declaration

Methods are inline by default
(without need for the inline modifier)

Separate implementation

```
class Fraction {
    public:
        int num() const;
        void num(int numerator);
        int den() const;
        void den(int denominator);

    private:
         int m_numerator {0}, m_denominator {1};
};
```

Definition (header file)

```
int Fraction::num() const {
        return m_numerator;
}
void Fraction::num(int numerator) {
        m_numerator = numerator;
}
int Fraction::den() const {
        return m_denominator;
}
void Fraction::den(int denominator) {
        m_denominator = denominator;
}
```

Implementation

:: is the scope resolution operator

Constructing an object

- The **constructor** method is used to initialize the fields of a class (a class can have multiple constructors)
  - If no constructor is defined a default one (taking no parameters) is generated by the compiler

```
ClassName::ClassName ( parameters )
: init-list
{
   body
}
```

optional

Member initialization list

The **member initialization list** (init-list) is used to initialize the fields of a class

- Each element of the comma-separated list has the following form:

```
membername { value }
```

Example

```
class Fraction {
    public:
        Fraction() : m_numerator{0}, m_denominator{1} {};
        Fraction(int numerator, int denominator=1)
                        : m_numerator{numerator},
                          m_denominator{denominator} {};

        int num() const { return m_numerator; }
        void num(int numerator) {
                        m_numerator = numerator; };
                int den() const { return m_denominator; }
        void den(int denominator) {
                        m_denominator = denominator; };

    private:
        int m_numerator, m_denominator;
};
```

Delegating constructors

- With the init-list we can chain constructors (*delegating constructors*)

```
class Fraction {
    public:
        Fraction() : Fraction{0,1} {};

        Fraction(int numerator, int denominator=1)
            : m_numerator{numerator},
              m_denominator{denominator} {
        };

    private:
        int m_numerator, m_denominator;
};
```

Calling the other constructor

Delegating constructors

- An initializer for a delegating constructor must appear alone

Error! The object has already been created

```
class Fraction {
    public:
        Fraction() : Fraction{0,1}, m_dato{3}, m_numerator{0} {};

        Fraction(int numerator, int denominator=1)
            : m_numerator{numerator},
              m_denominator{denominator} {
        };

    private:
        int m_numerator, m_denominator, m_dato;
};
```

Instancing an object

- When an object is instanced the constructor is called
- Objects can be instanced on the stack or on the heap
  - When allocating on the heap (with **`new`**) remember to free the memory with **`delete`**

```
Fraction f1; //  0/1
Fraction f2 {1, 2}; // 1/2
Fraction f3 {7, 5}; // 7/5
Fraction* f4{new Fraction; // 0/1
Fraction* f5{new Fraction{2,3}}; // 2/3
Fraction* f6{new Fraction(8}}; // 8/1
// ...
delete f4;
delete f5;
delete f6;
```

Referencing the current object

- Inside a methods we can refer to the object with **this**, which is a pointer

```
class Fraction {
    public:
        Fraction() : m_numerator{0}, m_denominator{1} {};
        Fraction(int numerator, int denominator=1)
                        : m_numerator{numerator},
                          m_denominator{denominator} {};

        int num() const { return this->m_numerator; }
        void num(int numerator) {
                  this->m_numerator = numerator; };
        int den() const { return this->m_denominator; }
        void den(int denominator) {
                  this->m_denominator = denominator; };

    private:
        int m_numerator, m_denominator;
};
```

friend

- Sometimes we need to bypass access rules and allow a function or class to access private fields
  - We can use **`friend`**

```
friend class TrustedClass;
friend int read(MyClass& c);
friend void Test::look(MyClass& c);
```

Example

```
#include <iostream>

using namespace std;

class Fraction {

        friend void curious(Fraction&);

    public:
        Fraction() : m_numerator{0}, m_denominator{1} {};
        Fraction(int numerator, int denominator=1)
                        : m_numerator{numerator}, m_denominator{denominator} {};

               int num() const { return m_numerator; }
        void num(int numerator) { m_numerator = numerator; };
               int den() const { return m_denominator; }
        void den(int denominator) { m_denominator = denominator; };

    private:
        int m_numerator, m_denominator;
};

void curious(Fraction& f)
{
        cout << f.m_numerator << "/" << f.m_denominator << endl;
}

int main()
{
        Fraction fr {1, 2}; // 1/2

        curious(fr);
}
```

Declaring the function as a friend

Can access private fields