Why C++ ?

Guillaume "Vermeille" Sanchez

April 11, 2012

Could we define C++ ?

"C++ is a light-weigth abstraction programming language" (Bjarne Stroustup)

Technical points

- Native code
- Portable
- Free
- Statically typed
- Compatible with C
- No garbage collection
- Metaprogramming

C++ is fast

"C++ is clean, safe and fast" (Herb Sutter)

- Native code
- Efficient standard library
- Pay (only) for what you use
- Templates

C++ is clean

"C++ is clean, safe and fast" (Herb Sutter)

- Imperative style
- Object Oriented
- Functionnal (yes)
- Overloading
- Templates
- etc...

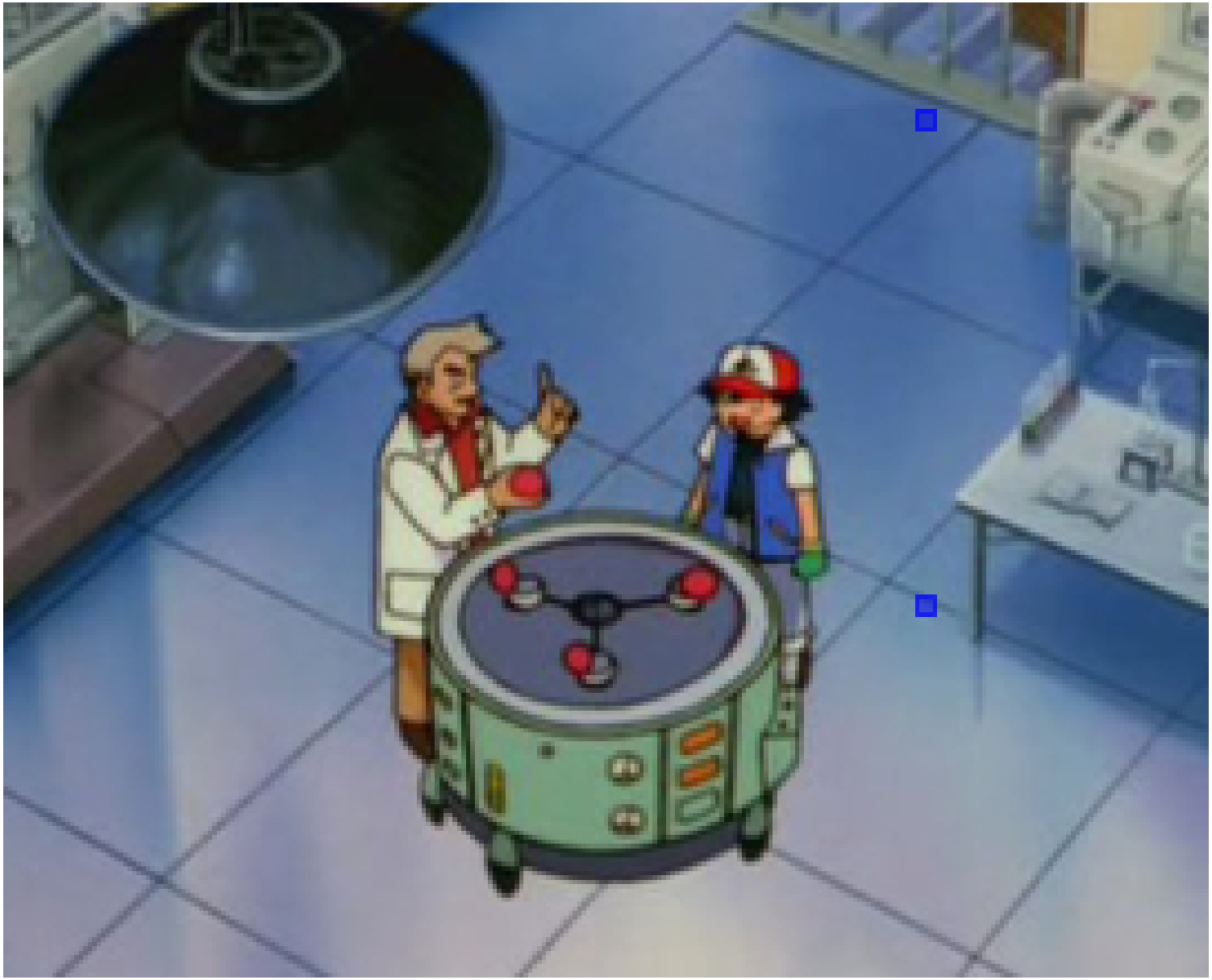
C++ is safe

"C++ is clean, safe and fast" (Herb Sutter)

- Memory safe (But we have pointers ?!)
- Thread safe
- etc...

Introduction

Let's start : a (very) quick point on C++ rules and syntax



Resolution operators

```
// :: namespaces and class static members
std::cout << "Hello World!" << std::endl;
MyClass instance;
// . instance's members, dereferencing references
instance.Func();
MyClass* pInst = &instance;
// -> or * for derefencing pointers
pInst->Func();
(*pInst).Func();
```

Scoping and deallocation

Same rule as in C applies : lexical scoping. As in C, dynamic allocation and static allocation are differentiated.

```
// Simple example
int Func(int a, int b)
{
    MyClass* bad_idea = new MyClass(b);

    if (a == 666)
    {
        MyClass better_idea(b);
        DoSomething(&better_idea);
        return 42; // memory leak
    } // better_idea is destructed HERE

    bad_idea->Hello();
    delete bad_idea;
    return 1337;
}
```

Pointers and references

```
int WithPointers(int* a)
{
    *a = 12; // dereferencing the pointer
    a += 3; // increments the pointer, certainly a mistake
    return *a + 6;
}

int WithReferences(int& a)
{
    a = 12; // syntax is the SAME for a non-reference
    a += 3;
    return a + 6;
}
```

Objects

```
// in myclass.h
class MyClass : public MyBaseClass
{
    public:
        MyClass();
        MyClass(const MyClass& other);
        int Test(void);
        int GetMember(void) const;
        const Param& GetParam(void) const;
    private:
        int member_;
        Param param_;
};
```

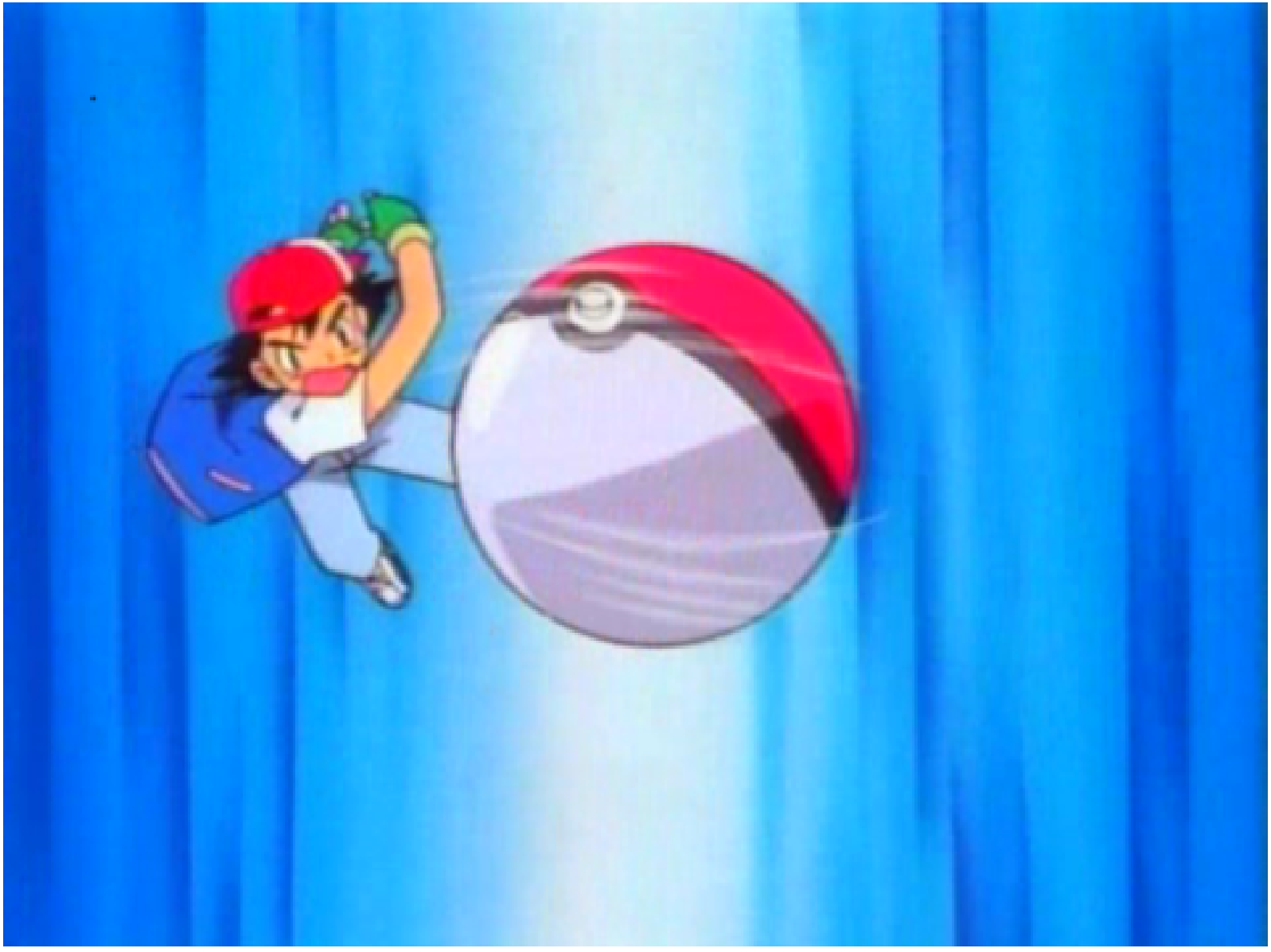
Objects

```
// in myclass.cpp
#include "myclass.h"
MyClass::MyClass()
    : member_(42), param_(666)
{
    ...
}
...
int MyClass::GetMember(void) const
{
    return member_;
}

const Param& MyClass::GetParam(void) const
{
    return param_;
}
```

Pokeball, Goooo !

We are now ready for serious code.



RAII : Resource Acquisition Is Initialization

```
void SaveInFile(const Options& op)
{
    FILE* file = std::fopen("options.xml");

    if (op.Number() == 666)
        throw std::exception("SATAN IS THERE"); // leak

    ...

    if (op.IsSpecial())
    {
        ...
        std::fclose(file); // Never Forget it !
        return;
    }

    std::fclose(file); // Never forget it !
}
```

- `std::fopen()`

  looks like a constructor

- `std::fclose()`

  looks like a destructor

RAII : Resource Acquisition Is Initialization

```
class FileHandler
{
    public:
        FileHandler(const std::string& path);
        ~FileHandler();
        bool IsOpened(void);
        void Append(const std::string& text);
        ...etc...
    private:
        FILE* file_pointer_;
};

FileHandler::FileHandler(const std::string& path)
{
    file_pointer_ = std::fopen(path.c_str());
}

FileHandler::~FileHandler()
{
    std::fclose(file_pointer_);
}

bool FileHandler::IsOpened(void)
{
    return file_pointer_ != 0;
}
```

RAII : Resource Acquisition Is Initialization

```
void SaveInFile(const Options& op)
{
    FileHandler file("options.xml");

    if (op.Number() == 666)
        throw std::exception("SATAN IS THERE"); // NO leak !

    ...

    if (op.IsSpecial())
    {
        ...
        return; // NO fclose !
    }

    // NO fclose again !
}
```

RAII adds safety and readability, very powerful C++ idiom. Not reproductible in C# : in C# you don't know when destructor is called.
ZERO overheads if FileHandler's function are inlined.
std::fstream does that.

std::unique_ptr

```
template <typename T>
class UniquePtr
{
    public:
        UniquePtr(T* ptr)
            : pointer_(ptr)
        {
        }
        ~UniquePtr()
        {
            if (pointer)
                delete pointer_;
        }
        T* operator->()
        {
            return pointer_;
        }
        T& operator*()
        {
            return *pointer_;
        }
    private:
        UniquerPtr(const UniquePtr&);
        UniquePtr& operator=(const UniquePtr&);
        T* pointer_;
};
```

A simple BinaryTree

```
template <typename T>
class BinaryTree
{
    public:
        T element;
        UniquePtr<BinaryTree<T>> left;
        UniquePtr<BinaryTree<T>> right;
};
```

That's all. No delete anymore.
(std::unique_ptr is a C++11 feature)

Templates

Templates

Templates are not :

- C# generics
- void*

But templates are :

- Compile-time calculation and code generation
- Metaprogramming
- Pattern matching
- Wonderful
- A gate to a functionnal world
- Work for your compiler

Generics

```
template <typename T>
class List
{
    public:
        List();
        T GetElem(void);

        ...operations...

    private:
        T elem_;
        std::unique_ptr<T> next_;
};

int main()
{
    List<int> list;
    list.Insert(42);
}
```

int parametrized templates

```
template <typename T, int Size>
class Array
{
    public:
        T& operator[](int i)
    {
        return array_[i];
    }
    ...etc...
    private:
        T array_[Size];
};

// Following types are *NOT* the same
Array<double, 7> seven;
Array<double, 6> six;
```

You can parametrize template with types (like in previous example) and with any integer based value : int, char, enum.

Structural & compile-time polymorphism

```
template <typename Object>
void SaveInDatabase(const Object& obj)
{
    Database db;
    obj.persist(db);
    /*
     * Object just needs to define Object::persist(Database& db);
     * No inheritance from, say, DBSerializable needed
     */
}

template <typename Fun>
void DoSomethingWithFourtyTwo(Fun f)
{
    f(42);
    /*
     * f is a function wich must take an int as its unique parameter
     * That's all. Templates allow to deal with functions like
     * functionnal languages do.
     */
}
```

Type Checking

Type Checking

C++ has robust type checking. Use it to generate compile-time errors.

```
template <typename T, int N, int M>
class Matrix
{
    public:
        Matrix(T initial)
        {
            for (int i = 0 ; i < N ; ++i)
                for (int j = 0 ; j < M ; ++j)
                    array_[i*N+j] = initial;
        }
        T& At(int line, int raw)
        {
            return array_[line*N+raw];
        }

        void operator+=(const Matrix<N, M>& other)
        {
            for (int i = 0 ; i < N ; ++i)
                for (int j = 0 ; j < M ; ++j)
                    array_[i*N+j] += other.array_[i*N+j];
        }
    private:
        T array_[N*M];
};
```

Type Checking

```
int main(void)
{
    Matrix<3, 3> a(0);
    Matrix<3, 4> b(0);
    a += b;
    /*
     * Compile-time error :
     * There is no void Matrix<3, 3>::operator+=(const Matrix<3, 4>&)
     */
}
```

Type Checking

```
template <int M, int K, int S>
class Value
{
    public:
        Value(float val)
            : value_(val)
        {}

        Value<M, K, S>& operator+=(const Value<M, K, S>& val) {
            value_ += val.value_;
            return *this;
        }

    private:
        friend ...operators...
        float value_;
};
template <int M1, int m2, int K1, int K2, int S1, int S2>
Value<M1+M2, K1+K2, S1+S2> operator*(const Value<M1, K1, S1>& val1,
                                     const Value<M2, K2, S2>& val2)
{
    return val1.value_ * val2.value_;
}
...etc..
```

(Example from Bjarne Stroustrup)

Type checking

```
typedef Value<1, 0, 0> Meters;
typedef Value<0, 1, 0> Kilograms;
typedef Value<0, 0, 1> Seconds;
typedef Value<1, 0, -1> Speed;
typedef Value<1, 0, -2> Acceleration;

int main()
{
    Meters m(10);
    Seconds s(2);
    Acceleration acc = m / s; // compile-time error
    Speed speed = m / s; // Ok
    return 0;
}
```

A word on C++'s functionnal aspect

Function as regular values

```
template <typename Return, typename Fun, typename Arg>
class LateCall
{
    public:
        LateCall(Fun f, Arg a)
            : f_(f), arg_(a)
        { }

        Return operator()(void)
        {
            return f(arg);
        }
    private:
        Fun f_;
        Arg arg_;
};

LateCall<void> push_later(&std::vector<int>::push_back, 42);
...
push_later();
```

boost::bind and std::functional do a lot more. You can even do partial application, change parameters order, etc, etc.

Functions as regular values

```
void DoSomethingWithAFunction(void fun(int, int), int arg)
{
    fun(arg, 42);
}

...
DoSomethingWithAFunction(&FunctionWith2Parameters, 666);
```

Metaprogramming : C++'s magick unleashed

Rules for Metaprogramming

Let's code some basics metaprogramming.
Templates are not mutables values. You have to code it in a functionnal way.
if/else are forbidden. They are *imperative* tests. You *must* use the ternary operator (condition) ? value1 : value2, exactly the same as the OCaml's if :

```
1 + (if condition then 1 else 2)
```

In C++ :

```
1 + ((condition) ? 1 : 2);
```

The compiler needs to have access to the value. Store it into a static const variable or into an enum.

```
template <int N>
struct Sum
{
    enum { value = (N == 0) ? 0 : N + Sum<N-1>::value; }
};
...
std::cout << Sum<45>::value << std::endl;
```

Specialization

There is another way to do the test : using specialization and pattern matching.

```
template <int N>
struct Sum
{
    enum { value = N + Sum<N-1>::value; }
};

template <>
struct Sum<0>
{
    enum { value = 0; }
};

/*
 * constexpr is C++11 feature that tells the compiler to try to compute the
 * value if parameters are known at compile time.
 */

constexpr int Sum(int n)
{
    return n ? (n+Sum(n-1)) : 0;
}
...
std::cout << Sum(42) << std::endl;
```

Partial Specialization

```
template <int N>
struct Number
{
    enum { value = N };
};

template <bool Cond, typename Yes, typename No>
struct If;

template <typename Yes, typename No>
struct If<true, Yes, No>
{
    enum { value = Yes::value };
};

template <typename Yes, typename No>
struct If<false, Yes, No>
{
    enum { value = No::value };
};

std::cout << If<1 == 2, Number<42>, Number<666> >::value << std::endl;
```

Metaprogramming : policies

```
template <typename T, typename <class> Container, typename OnEmpty>
class Stack : private Container<T>
{
    public:
        ...
        void Push(const T& x)
        {
            Container<T>::push_front();
        }

        T Pop(void)
        {
            if (!Container<T>::empty())
                return Container<T>::pop_front();
            else
                return OnEmpty::OnEmpty();
        }

        bool empty()
        {
            return Container<T>::empty();
        }
};

typedef Stack<int, MyLinkedList, ThrowWhenEmpty> IntStack;
```

The End.