The C++ language
and the standard library

By selected examples

Lectures on Modern Scientific Programming
Wigner RCP
23-25 November 2015

D. Berényi – M. F. Nagy-Egri

C++ language

Some design directives: (read more)

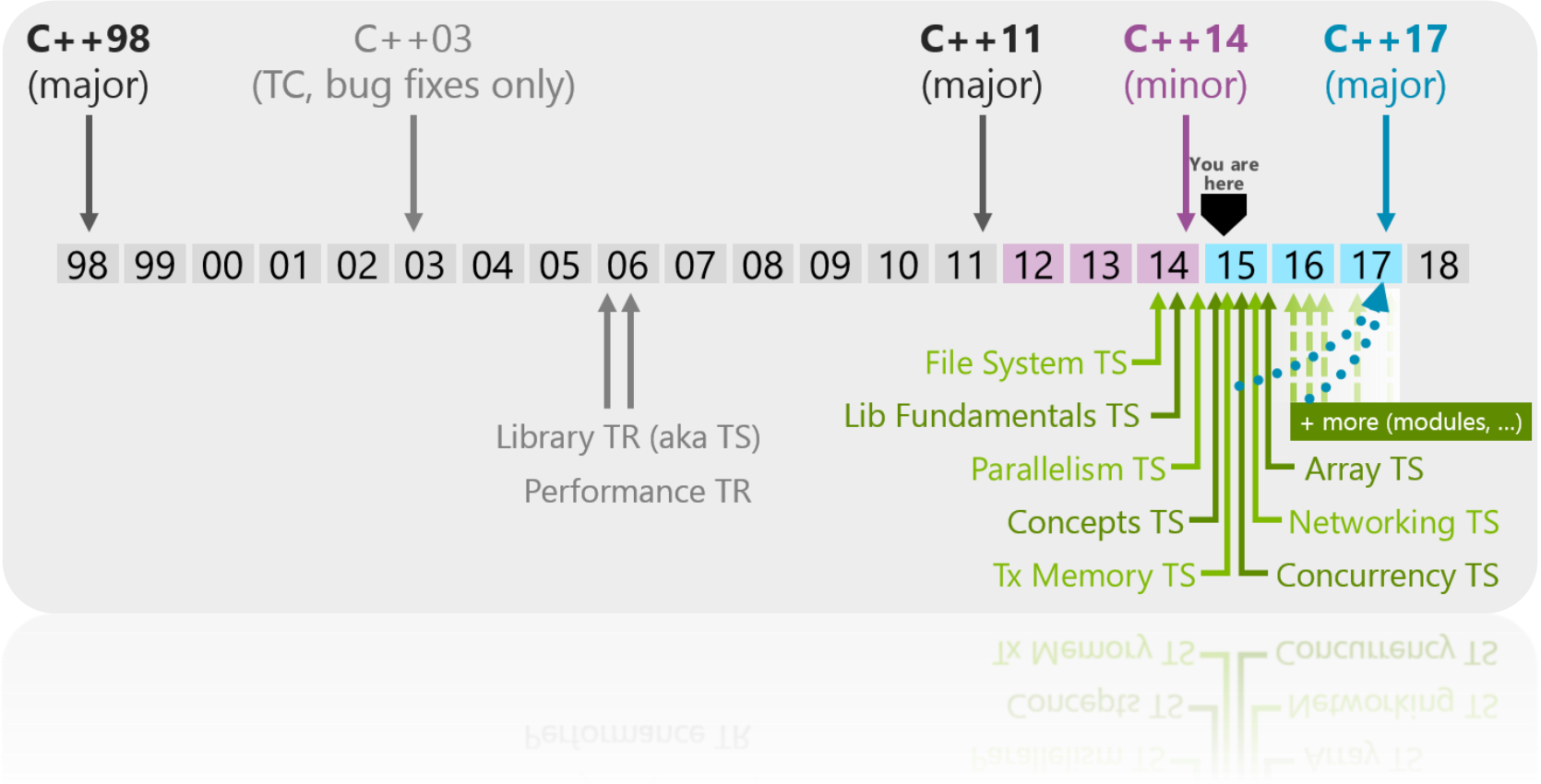
- A useful language for real world applications
- Do not force people into a specific programming style
- Direct mapping to hardware
- No implicit violations of the static type system
- Compatibility with C
- What you don't use, you don't pay for (zero-overhead rule)

Bjarne Stroustrup

A link to his presentation at the Eötvös University in 2014

C++ language

Evolution of standardization:



C++ language

Features:

- Imperative programming
- Object-oriented programming (classes, inheritance)
- Functional programming (esp. since C++11: lambda functions)
- Generic programming (templates)

C++ language

Available free compilers:

- Gnu Compiler Collection (GCC):

  Mostly used on Linux/Mac systems,
  Current version 5.2 C++14 feature complete.

- Clang (LLVM):

  The most rapidly developing and most standard compliant compiler platform
  Easy to build tools to it

  They are working on C++17 already.
  Drop-in compatible with GCC. Works on UNIX, Mac, almost fully working on windows

- Visual Studio (IDE) MSVC (Visual C++ compiler):

  The slowest developing compiler, specific to Windows (many non standard behaviours)
  (although a UNIX version is coming).

Proprietary

- Intel C++ compiler: best performance especially on Intel processors.

C++ language

Online compilers:

Useful for experimenting with language features and checking small programs

- gcc.godbolt — gcc up to 5.2, clang up to 3.7, icc 13.0.1, and more, cannot run program, but shows disassembly
- Ideone — gcc 5.1, can run the program
- Tutorialspoint — gcc 4.9.2, can compile multiple sources together
- vc++ webcompiler — latest nightly build of the msvc compiler

C++ language

Recommendations:

- Use clang!
  Most standard compliant, best error messages
- Try your code with at least two compilers
  why? compiler bugs, standard compliance
  GCC + clang on linux, msvc + clang on windows
- Use an IDE!
  Qt Creator on Linux, Visual Studio on Windows

C++ language

C++ is a very complex language... Hard to master it.

Here we do not go through all the language constructs in details, rather give some examples and links to references

Disclaimer

These examples are provided for demonstration purposes only.

They do not necessarily entirely correct for every imaginable input, and not expected to work in all and any circumstances, etc.

Even, if you play with them, you will run into more questions. But really, this is what you should do!☺

C++ language

If you have a question:

- google for it
- Check for the problem on stackoverflow.com
- Read the relevant parts of cppreference.com
- Ask us☺

C++ language by examples

All example codes are available on the webpage!

(TODO: insert link here!)

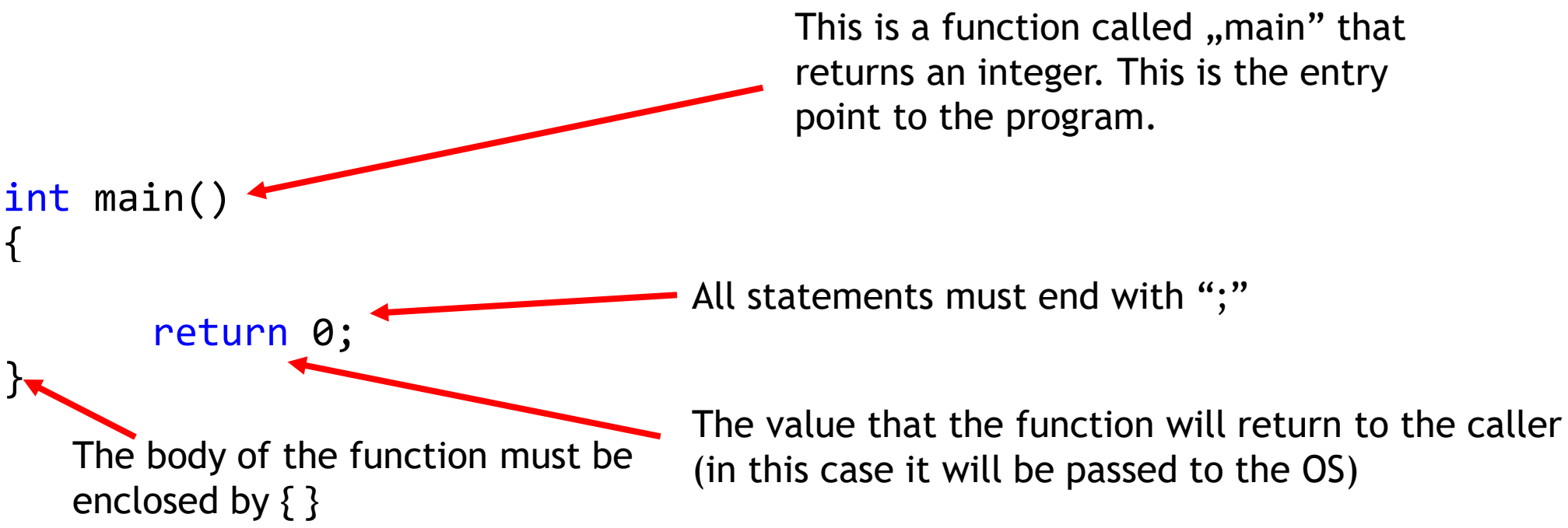
C++ language by examples

Simple hello world:

```
#include <iostream>

int main()
{
	std::cout << "Hello World" << std::endl;
	return 0;
}
```

C++ language by examples

Simple hello world:



C++ language by examples

Simple hello world:

This is the way to use code from other source files, this time the Input/Output library from the file „iostream”

std is a namespace, all names inside it can be accessed by `std::nameofsomething`

`std::cout` is the object representing the standard output

Character string to display

```
#include <iostream>

int main()
{
        std::cout << "Hello World" << std::endl;
        return 0;
}
```

The platform specific line end string and flush

C++ language by examples

A simple function:

```
double addSq(double x, double y)
{
    return x*x + y*y;
}
```

A simple function:

This is the name of the function

This is the type of the value that the function will return

Between the ( )s is the argument list of the function (type and name pairs, separated by commas)

```
double addSq(double x, double y)
{
      return x*x + y*y;
}
```

Body of the function

C++ language by examples

A simple function:

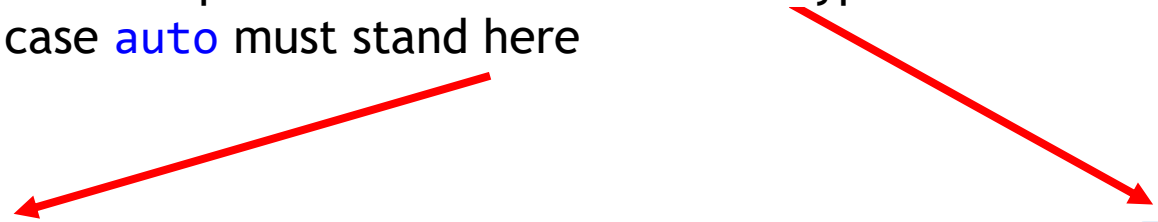
C++11 introduced the trailing return type,
in this case `auto` must stand here

Why?
Details later in the template metaprogramming session.

```
auto addSq(double x, double y)->double
{
    return x*x + y*y;
}
```

A simple function:

C++14 makes it possible to infer the return type of a function from the return statements!
in this case `auto` must stand here



```
auto addSq(double x, double y)
{
    return x*x + y*y;
}
```

Nothing is here

C++ language by examples

Some simple array manipulation, traditional C:

```
int values[3];
for(int i=0; i<3; i=i+1)
{
	values[i] = i+1;
}

int sum = 0;
for(int i=0; i<3; i=i+1)
{
	sum = sum + values[i] * values[i];
}
```

Some simple array manipulation, traditional C:

```
int values[3];
for(int i=0; i<3; i=i+1)
{
        values[i] = i+1;
}

int sum = 0;
for(int i=0; i<3; i=i+1)
{
        sum = sum + values[i] * values[i];
}
```

An array that stores 3 integers

A loop statement, that declares a loop variable (i), initializes it to 0 and repeats the statements inside { } with i being increased by one each time until i<3 holds.

Equivalent to:
values[0] = 1;
values[1] = 2;
values[2] = 3;

This loop calculates the sum of squares of the array into the variable sum

C++ language by examples

While loops are flexible and powerful,
they are prone to indexing (and scoping) errors.

Try to avoid explicit loops and use algorithms instead!

C++ language by examples

```
#include <array>    //for std::array
#include <numeric>  //for std::iota, std::accumulate
#include <iostream> //for std::cout

int main()
{
	std::array<int, 5> values;

	std::iota( values.begin(), values.end(), 1);

	auto sum = std::accumulate( values.begin(),
	                            values.end(),
	                            0,
	[](int a, int b){ return a+b*b; } );

	std::cout << "Sum is: " << sum << std::endl;
	return 0;
}
```

C++ language by examples

```
#include <array>    //for std::array
#include <numeric>  //for std::iota, std::accumulate
#include <iostream> //for std::cout

int main()
{
        std::array<int, 5> values;

        std::iota( values.begin(), values.end(), 1);

        auto sum = std::accumulate( values.begin(),
                                    values.end(),
                                    0,
        [](int a, int b){ return a+b*b; } );

        std::cout << "Sum is: " << sum << std::endl;
        return 0;
}
```

An array that stores 5 integers

Start and end of the std::array

std::iota is a function that fills an array starting from the value of the last argument (now: 1), and increasing it at each step.

C++ language by examples

```
#include <array>     //for std::array
#include <numeric>   //for std::iota, std::accumulate
#include <iostream> //for std::cout

int main()
{
        std::array<int, 5> values;

        std::iota( values.begin(), values.end(), 1);

        auto sum = std::accumulate( values.begin(),
                                    values.end(),
                                    0,

        [](int a, int b){ return a+b*b; } );

        std::cout << "Sum is: " << sum << std::endl;
        return 0;
}
```

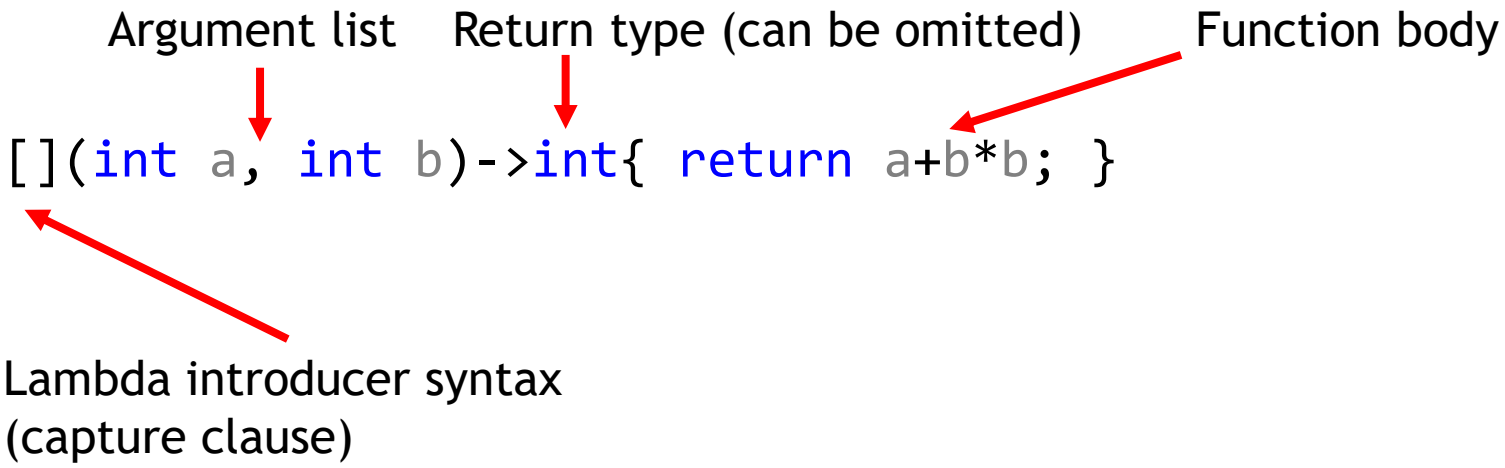
Automatically inferred type (C++11)
(will be `int` because 0 here is `int`)

`std::accumulate` is a function that calculates a reduction of a sequence of elements

C++ language by examples

C++11 introduced the lambda functions (anonymous functions).

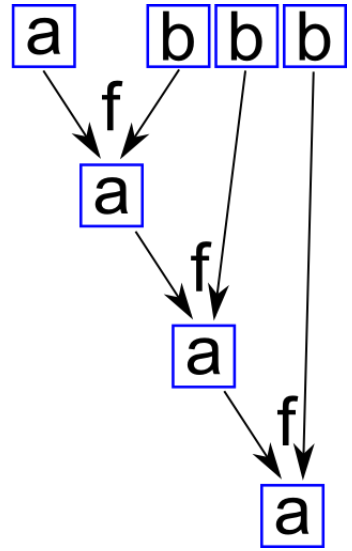
These are expressions(!), that are in-place declared name-less functions:



C++ language by examples

```
#include <array>    //for std::array
#include <numeric>  //for std::iota, std::accumulate
#include <iostream> //for std::cout

int main()
{
        std::array<int, 5> values;

        std::iota( values.begin(), values.end(), 1);

        auto sum = std::accumulate( values.begin(),
                                    values.end(),
                                    0,
        [](int a, int b){ return a+b*b; } );

        std::cout << "Sum is: " << sum << std::endl;
        return 0;
}
```

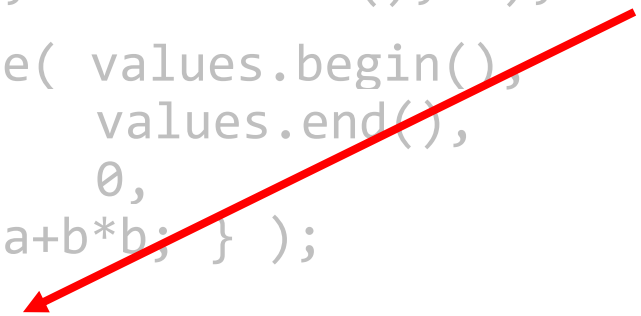
`std::accumulate` use this lambda function to reduce the elements of the array, by repeatedly applying it:



C++ language by examples

```
#include <array>    //for std::array
#include <numeric>  //for std::iota, std::accumulate
#include <iostream> //for std::cout

int main()
{
        std::array<int, 5> values;

        std::iota( values.begin(), values.end(), 1);

        auto sum = std::accumulate( values.begin(),
                                    values.end(),
                                    0,
        [](int a, int b){ return a+b*b; } );

        std::cout << "Sum is: " << sum << std::endl;
        return 0;
}
```

Write out the string "Sum is: " followed immediately by the value of the variable sum and a line break.



C++ language by examples

```
#include <array>    //for std::array
#include <numeric>  //for std::iota, std::accumulate
#include <iostream> //for std::cout

int main()
{
	std::array<int, 5> values;

	std::iota( values.begin(), values.end(), 1);

	auto sum = std::accumulate( values.begin(),
	                            values.end(),
	                            0,
	[](int a, int b){ return a+b*b; } );

	std::cout << "Sum is: " << sum << std::endl;
	return 0;
}
```

C++ language by examples

When using the simplest argument passing, a local copy of the value is created for the function:

These are local variables, containing a copy of the values from the call site!

```
auto addSq(double x, double y)
```

C++ language by examples

If we have a large object, we may want to avoid the copy.

There are multiple choices:

```
auto f(BigObject* x)
```
Passing by naked pointer Not recommended!!!

```
auto f(std::shared_ptr<BigObject> x)
```
Passing by a smart pointer (C++11)
Involves some overhead

```
auto f(BigObject& x)
```
Passing by reference *recommended*

A reference is physically a pointer,
but its semantics are much more restricted (cannot be null, no arithmetic on it, etc.)

C++ language by examples

Value categories: (since C++11)

- L-value: any value that is ***bound*** to a name already
  Can be assigned to (can stand on the left of the operator=), it has address in memory
- Pure r-value: temporary, ***not bound*** to a name!
  Examples: a literal, or a value resulting from a function call
  No address in memory, can only stand on the right side of operator=
- X-value: eXpiring object, ***not bound*** to a name!
  No address in memory, can only stand on the right side of operator=
  Examples: function returning an r-value reference, `static_cast` to an r-value reference, member of an x-value object

- Gl-value: L-value + X-value -> they bind to `const&`
- R-value: X-value + Pure R-value -> they prefer `&&` over `const&`

C++ language by examples

C++11 introduced a new kind of reference, that is called the r-value reference.

```
auto f(BigObject* x)

auto f(std::shared_ptr<BigObject> x)

auto f(BigObject& x)

auto f(BigObject&& x)
```

This expresses, that the function will take ownership of the object being passed to it (it may only involve a shallow copy)

Call site:

```
BigObject bo = ...;

auto result = f( std::move(bo) );
auto result = f( makeBigObject(...) );
```

The same transfer may happen if the object is being passed as a return value from another function

C++ language by examples

Rule of thumb, what to use and when:

`auto f(Object x)`	When the object is small (say <=128 bit), and/or copy is needed
`auto f(Object* x)`	Never, unless you know exactly what and why you are doing
`auto f(std::shared_ptr<Object> x)`	When you really need to share the object with other parts of the program, and x may be null
`auto f(BigObject& x)`	When you really need to modify the outer object
`auto f(BigObject const& x)`	When you don't need to modify the object
`auto f(BigObject&& x)`	When you'd like to take ownership of the object

One more thing about references

The perfect forwarding problem:

Many times, you'd need to duplicate code, just because it should work both with `const`&s and &&s!
This is taken care by the language if the type is a template:

```
template<typename T>
auto foo( T&& t )
{
    return bar( std::forward<T>(t) );
}
```

`bar` will receive the value as it was passed to `foo`:
if it was a `const&`, it'll bind to `bar(const&)`,
if it was a `&&`, it'll bind to `bar(T&&)`

C++ language by examples

Structs may be familiar to C programmers:

- Structs are collections of variables and functions that manipulate them
- They are the building blocks of object-oriented programming

C++ language by examples

Classes extend the functionality of structs

- Classes may enforce access-control on members (both data and functions)
  - A struct is a class with all it's members being public

In C++, both Classes and Structs may

- Inherit from one another (is-a, has-a relationship)
- Have virtual functions (will not treat it here)

Objects:
introduce a new type, that encapsulates data and functions

```
struct Circle
{
    double r;

    double area(){ return r*r*3.1415; }
};
```

Objects:
introduce a new type, that encapsulates data and functions

keyword: struct or class

Name of the new type

Data members:
type and name pairs, separated by ";"

Member functions

```
struct Circle
{
    double r;

    double area(){ return r*r*3.1415; }
};
```

Such an object can be instantiated on the *stack* and its members can be accessed by ".":

```
Circle c{2.};
auto R = c.r;
auto A = c.area();
```

Such an object can be instantiated on the *heap* and its members can be accessed by “->”:

Here, we use the smart pointer std::shared_ptr that manages deallocation.

Avoid using naked `new`, it is much easier to make a memory management mistake with it.

```
#include <memory>

std::shared_ptr<Circle> c =
                        std::make_shared<Circle>(Circle{2.});
auto R = c->r;
auto A = c->area();
```

C++ language by examples

Constructors and destructors:

Objects have special functions that manage their creation and disposal.

Constructors create a new instance of the object,
Default cons. (takes no argument), Copy cons., Move cons.

The destructor finalizes the state of the object for removal from memory.

C++ language by examples

Constructors and destructors are the quintessence of C++

The language makes very strong guarantees on running constructors and destructors and their ordering

These guarantees drive the RAII idiom in C++

- This is the feature used by smart pointers and mutex locks (see later)

This is why C++ does not need garbage collection and why it's fast

Constructors and destructors:

Until you only use built in types and library objects, you don't need to worry about construction and destruction and assignment, the compiler will generate the necessary functions automatically:

```
struct Circle
{
      std::array<double, 2> position;
      double r;
};
Circle c{ {{2., 5.}}, 2.5 };
Circle c2; c2 = c;
```

Here we use C++11 list initialization

C++ language by examples

Constructors and destructors:
If you need to do something more involved (e.g. has ownership) read the rule of three/five/zero:

```
struct Vector
{
    size_t n;
    std::unique_ptr<T[]> data;
}
```

C++ language by examples

Constructors and destructors:
In the most generic case, you need to provide the following:

```
struct Vector
{
        Vector()
        Vector( Vector const& cpy );
        Vector( Vector && mv );

        Vector& operator=( Vector const& cpy );
        Vector& operator=( Vector && mv );

        ~Vector();
}
```

Default constructor

Copy constructor

Move constructor

Copy assignment

Move assignment

Destructor

C++ language by examples

About doing mathematics in C++:

C++ has:

- Built in numeric types (int, float, double) with arithmetic operations
- Has common and special functions (see the complete list here, it got extended in C++11)
- Has complex data type, with arithmetic and trigonometric support (augmented in C++11)
- Compile time rational arithmetic (since C++11)

C++ language by examples

About doing mathematics in C++:

C++ has:

- A type named valarray for array-wise operations (C++11)
  (although the design is flawed a bit, it is not very recommended)
- A pseudo-random number library (C++11)
  Designed by a physicist☺: several good generators and several widely used distributions
- Few widely used algorithms:
  std::iota, std::accumulate, std::inner_product, std::adjacent_difference, std::partial_sum

C++ language by examples

About doing mathematics in C++:

What C++ is badly missing:

- A standard n-dimensional Vector, Matrix class and linear algebra on it
  It is just plain too hard for the committee to agree on the design of such a big beast…

C++ language by examples

Some other examples...

C++ language by examples

Functions can return only one object.

But that can be composite: use std::tuple

```
std::tuple<int, int, double> f(int a, int b)
{
	return std::make_tuple(a, b, (double)a/(double)b);
}
```

std::tie can be used to expand the contents of the tuple into l-value references!

C++ language by examples

Do not write algorithms always by hand, always check if it is available in the standard library:

- Non-modifying sequence operations
- Modifying sequence operations
- Partitioning
- Sorting
- Binary search
- And more...

Many of these got extended in C++11/14!

Using them is simple:

```
std::array<int, 7> values{11, 42, 2, 56, 7, -1, 6};
std::sort(values.begin(), values.end());
```

C++ language by examples

Also check out the containers library:

- Sequence containers
  (O(1) and O(N))
- Associative containers
  (key based O(log N) access)
- Unordered associative containers (C++11)
  (key based, hashed with O(1) amortized access)
- Adaptors

Many of these got revised / extended in C++11/14!

C++ language by examples

I/O and strings: use iterators and streams!

We give examples to read manipulate and write data

Stream:
An object, that represents sequence of characters or binary data (most prominently: files and console)

Iterator:
An object, that represents position in a container or stream
(generalization of a pointer or index)

C++ language by examples

Opening a file and reading ints from it:

```
std::vector<int> data;


std::ifstream input("data.txt", std::ios::in);
if( !input.is_open() )
{
      std::cout << "Could not open input file " << std::endl;
}
std::copy( std::istream_iterator<int>(input),
           std::istream_iterator<int>(),
           std::back_inserter(data) );
```

C++ language by examples

Opening a file and reading ints from it (separated by whitespace):

```
std::vector<int> data;


std::ifstream input("data.txt", std::ios::in);
if( !input.is_open() )
{
      std::cout << "Could not open input file " << std::endl;
}
std::copy( std::istream_iterator<int>(input),
           std::istream_iterator<int>(),
           std::back_inserter(data) );
```

Input file stream object

Check if file could be located and opened

Begin of file iterator

End of file iterator

Iterator to insert into data

Opening a file and writing ints to it (separated by whitespace):

```
std::vector<int> data;

std::ofstream output("data2.txt", std::ios::out);

std::copy( data.begin(),
           data.end(),
           std::ostream_iterator<int>(output, " ") );
```

Output file object

Begin of container

End of container

Iterator to output file, separator is space

Streams provide a formatted output operator:

```
std::ifstream file("data.txt", std::ios::out);

int i = 0;
file >> i;
```

Converts characters into an integer automatically if possible.

If it fails, `i` is unmodified!

C++ language by examples

You can create a stream of characters and use it as an universal converter from/to strings:

```
std::stringstream ss;
double x = sqrt(2.0);
ss << x;
```

Converts the floating point value to string in ss

```
std::string s = ss.str();
```

```
float y = 0.0f;
ss >> y;
```

Converts the string back to a floating point value

```
if( ss ){ std::cout << "Conversion to float succeeded." << std::endl; }
else    { std::cout << "Conversion to float failed." << std::endl; }
```

C++ language by examples

What are these “>>” and “<<“ really?

Will they work for user defined types?

A user defined data type:

```
struct Data
{
    int i;
    double x;
    std::string s;
};
```

C++ object for representing ASCII strings with some useful methods

C++ language by examples

A user defined data type and the stream out operator implementation for it:

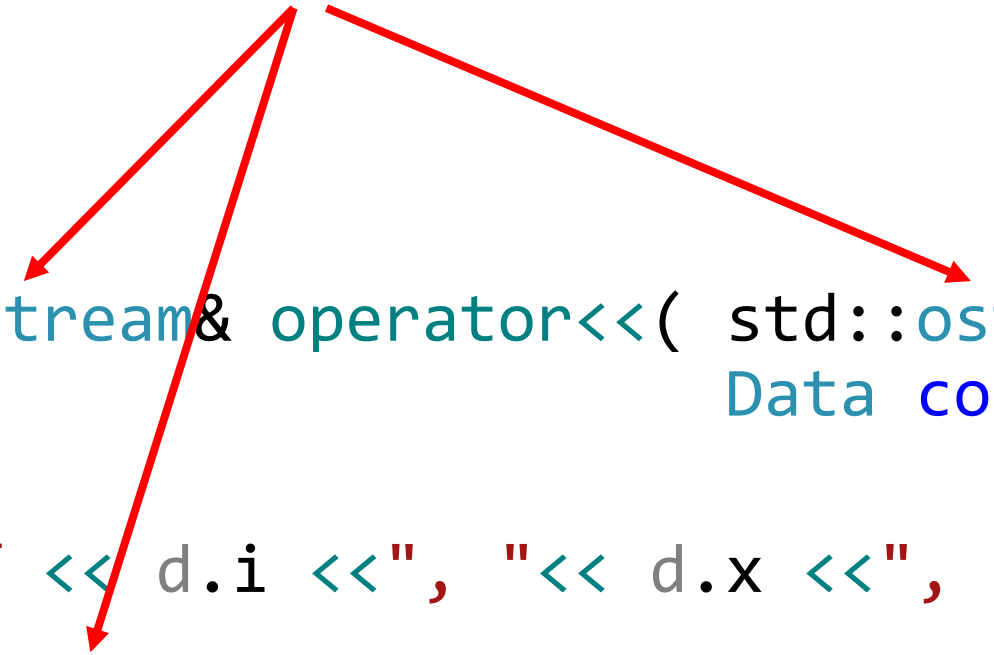
```
struct Data{ int i; double x; std::string s; };
```

C++ operators are just functions where name the is `operator` keyword + symbol

```
std::ostream& operator<<( std::ostream& s,
                          Data const  & d )
{
 s<<"{“ << d.i <<", "<< d.x <<", "<< d.s.c_str() << "}";
 return s;
}
```

C++ language by examples

C++ object for representing a generic output stream.
It has to be passed back as the return value



```
std::ostream& operator<<( std::ostream& s,
                          Data const  & d )
{
 s<<"{“ << d.i <<", "<< d.x <<", "<< d.s.c_str() << "}";
 return s;
}
```

C++ language by examples

Input:

```
std::istream& operator>>( std::istream& s, Data& d )
{
        std::string tmp;
        std::getline(s, tmp);
        if(tmp.size() > 0)
        {
            std::stringstream ss(tmp);
            std::getline(ss, tmp, ','); d.i = std::stoi(tmp);
            std::getline(ss, tmp, ','); d.x = std::stod(tmp);
            std::getline(ss, d.s);
         }
         return s;
}
```

C++ object for representing a generic input stream.
It has to be passed back as the return value

C++ language by examples

- Time measurement was standardized in C++11:

```
auto t1 = std::chrono::high_resolution_clock::now();

//some lengthy operation...

auto t2 = std::chrono::high_resolution_clock::now();

long long duration =

std::chrono::duration_cast<std::chrono::microseconds>(t2-t1).count();
```

64 bit signed integer type (C++11)

C++ language by examples

Very basic Vector type:

```
struct Vector2i{ int x, y; };


Vector2i operator+( Vector2i const& u, Vector2i const& v )
{
	return Vector2i{ u.x+v.x, u.y+v.y };
}

Vector2i operator-( Vector2i const& u, Vector2i const& v )
{
	return Vector2i{ u.x-v.x, u.y-v.y };
}
```

Very basic Vector type:

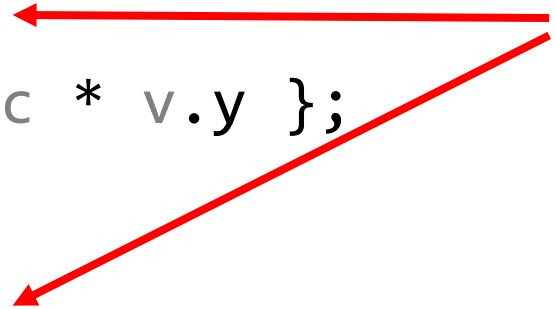
```
struct Vector2i{ int x, y; };
```

```
Vector2i operator*( int c, Vector2i const& v )
{
      return Vector2i{ c * v.x, c * v.y };
}
```

```
Vector2i operator*( Vector2i const& v, int c )
{
      return Vector2i{ c * v.x, c * v.y };
}
```

Operators are non-commutative!

We have to define both left and right scalar product:



C++ language by examples

Very basic Vector type:

```
struct Vector2i{ int x, y; };
```

We recommend to have the scalar product as a function instead of an operator. May lead to surprises otherwise.

```
int dot( Vector2i const& u, Vector2i const& v )
{
	return u.x * v.x + u.y * v.y;
}
std::ostream& operator<<( std::ostream& s, Vector2i const& v )
{
	s << "{" << v.x << ", " << v.y << "}";
	return s;
}
```

C++ language by examples

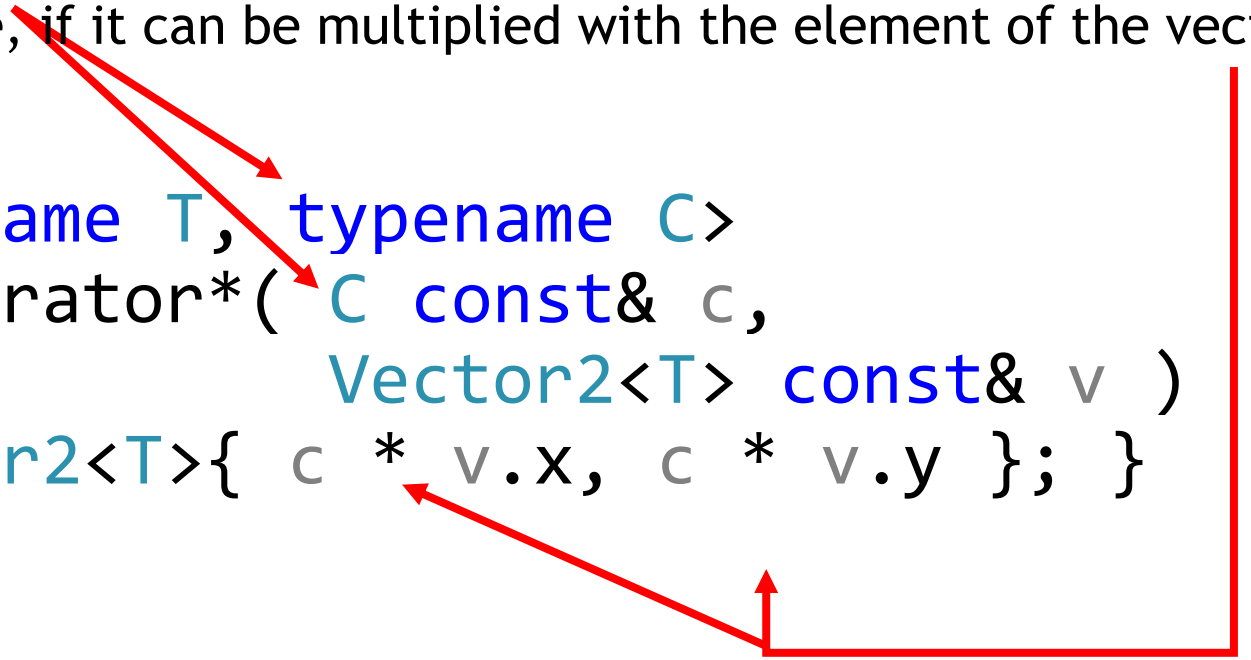
Templated Two-vector type,
it can work with any type, that has the proper operators defined:

```
template<typename T>
struct Vector2
{
    T x, y;
};
```

```
template<typename T>
Vector2<T> operator+( Vector2<T> const& u,
                      Vector2<T> const& v )
{ return Vector2<T>{ u.x+v.x, u.y+v.y }; }

template<typename T>
Vector2<T> operator-( Vector2<T> const& u,
                      Vector2<T> const& v )
{ return Vector2<T>{ u.x-v.x, u.y-v.y }; }
```

Note: the “scalar” type here, is not restricted to be the element type of the vector, rather, anything is fine, if it can be multiplied with the element of the vector!

```
template<typename T, typename C>
Vector2<T> operator*( C const& c,
                      Vector2<T> const& v )
{ return Vector2<T>{ c * v.x, c * v.y }; }
```



```
template<typename T, typename C>
Vector2<T> operator*( C const& c,
                      Vector2<T> const& v )
{ return Vector2<T>{ c * v.x, c * v.y }; }


Vector2<Vector2<int>> q{ {-1, 1}, {3, 6} };
Vector2<Vector2<int>> r{ {1, -2}, {5, 2} };

3 * q + r
```

So code like this does what we expect!

C++ language by examples

Writing a nice and capable Vector class needs multiple building blocks, we show some things to be aware of...

Some samples will be available on the webpage!

How to store the elements?

```
template<typename T>
struct Vector
{
    size_t n;
    T*     data;
    Vector(size_t sz):n(sz), data(new T[n]){}
    ~Vector{ delete[] data; }
};
```

Naked pointer, minimal size, fastest, you are responsible for everything!

new[],

delete[] Only use them if you are sure!

How to store the elements?

```
template<typename T>
struct Vector
{
    std::vector<T> data;
    Vector(size_t sz):data(sz){}
    ~Vector = default;
};
```

Use a container, if you are lazy:)

and don't want to bother with memory management too much

C++11: explicit signal that the compiler generates the destructor

How to store the elements?

```
template<typename T>
struct Vector
{
  size_t n;
  std::unique_ptr<T[]> data;
  Vector(size_t sz):n(sz),
                    data(std::make_unique<T[]>(n)){}
  ~Vector = default;
};
```

C++11: Use a smart pointer.
Minimal overhead
Harder to make a mistake,
and takes care of the deallocation

C++11: explicit signal that the compiler generates the destructor

How to access the elements?

These must be inside the class definition:

```
struct Vector
{
   ......
  T&       operator[]( size_t i )       { ... }
  T const& operator[]( size_t i )const { ... }
};
```

`operator[]` can only take 1 argument! (language limitation)

This one will be used to write to the elements

This one will be used when only read is possible (on a `const` Vector)

How to make it compatible with the standard library?

```
struct Vector
{
   ......
   T*       begin(){ return data.get(); }
   T*       end()  { return data.get()+n; }
   T const* cbegin() const { return data.get(); }
   T const* cend()   const { return data.get()+n; }
};
```

begin should point to the first element

end should point ***after*** the last element

`const` versions for reading only!

C++ language by examples

- Notes: this does only work because naked pointers treated specially in the standard library, and they by definition fulfil the requirements of RandomAccessIterator

  (In C++17, it will fulfil the even stronger ContiguousIterator concept)

  Read more to see what is required to write a valid iterator object.

C++ language by examples

- Problem:

  The whole iterator based design is outdated...

  This is known, the designers of the language started working on STL2 and the proposed ranges are going to remedy the situation…

  Want something more composable? Check out the afternoon session...

C++ language by examples

- Some counter examples:

```
template<typename T>
Vector<T> operator+( Vector<T> const& u, Vector<T> const& v )
{
    assert( u.size() == v.size() );
    Vector<T> result( u.size() );
    std::transform( u.cbegin(), u.cend(), v.cbegin(),
                    result.begin(),
                    [](T const& uu, T const& vv){ return uu + vv; } );
    return result;
}
```

Potentially uninitialized memory exists here, the object will be initialized externally...

STL algorithms cannot create objects, they can only write to them...

C++ language by examples

- Some counter examples: a simple integrator

```
template<typename F, typename T>
auto TrapezoidIntegrator1(size_t n, F const& f, T const& x0, T const& x1)
{
   double sum = 0.0;
   double dx = (x1 - x0) / (double)n;
   for (size_t i = 0; i<n; i++)
   {
      sum += f(x0 + dx*(double)i);
   }
   return sum * dx;
};
```

C++ language by examples

- Some counter examples: same integrator in STL

```
template<typename F, typename T>
auto TrapezoidIntegrator2(size_t n, F const& f, T const& x0, T const& x1)
{
   double dx = (x1 - x0) / (double)n;
   size_t i = 0;

   std::vector<T> values(n);
   std::generate(values.begin(), values.end(),
         [=, &i]{ auto res = f(x0 + dx * (double)i); ++i; return res; });
   return dx * std::accumulate( values.begin(), values.end(), (T)0,
                                [](T const& a, T const& b){ return a+b; } );
};
```

C++ language by examples

- Some counter examples: same integrator in STL

```
template<typename F, typename T>
auto TrapezoidIntegrator2(size_t n, F const& f, T const& x0, T const& x1)
{
    double dx = (x1 - x0) / (double)n;
    size_t i = 0;

    std::vector<T> values(n);
    std::generate(values.begin(), values.end(),
          [=, &i]{ auto res = f(x0 + dx * (double)i); ++i; return res; });
    return dx * std::accumulate( values.begin(), values.end(), (T)0,
                                 [](T const& a, T const& b){ return a+b; } );
};
```

We need this temporary to use the algorithms!

C++ language by examples

- Some counter examples: simple integrator native vs STL:

  The difference in this case (200000 points):

Compiler	Naïve implementation	STL idiomatic implementation
MSVC	6 ms	8 ms
clang (windows)	10.5 ms	12.5 ms

C++ language by examples

STL Threading:

Multi-threading became standard with C++11!

With all syncronisation and atomic operation primitives!

C++ language by examples

Here we only review one of the simplest-to-use primitive:

```
std::future<R> handle =
std::async(std::launch::async, [](...){...}, ...);
```

Here we only review one of the simplest-to-use primitive:

This signals, that we want to run our new thread separately from this current one

This lambda is the function, that will execute in the new thread

These arguments will be passed to the lambda when the thread starts

```
std::future<R> handle =
std::async(std::launch::async, [](...){...}, ...);
```

C++ language by examples

Here we only review one of the simplest-to-use primitive:

R should be the return value of the lambda

std::future is an object that ***will*** hold the result of the thread ***when*** it is finshed!

```
std::future<R> handle =
std::async(std::launch::async, [](...){...}, ...);
```

Here we only review one of the simplest-to-use primitive:

```
std::future<int> handle =
std::async(std::launch::async,
           [](int a, int b){ return a+b; }, 2, 5 );
//do stuff
int result = handle.get();
```

`.get()` will only return when the thread is finished!

Here we only review one of the simplest-to-use primitive:

You can create an array of futures, and launch multiple thread in a loop.

Than wait for them in a loop. See the muti-threaded integrator example.

```
unsigned int nMaxThreads =
std::thread::hardware_concurrency();
```

This gives you the maximum number of threads that your computer can run simultaneously.

C++ language by examples

Atomic operations support library (C++11)

Atomics are operations that are guaranteed that they will not be interrupted by other threads.

I.e. safe to use in multithreading environments for a lockless sharing of resources.

C++ language by examples

Atomic operations support library (C++11)

You can make any type atomic, but it will be only efficient for built in types, that are supported by the hw and OS.

Initialization and many operations are supported, like load, exchange, add/sub/and/or/xor

C++ language by examples

Regular expressions library (C++11):

Need the functionality of awk / grep for finding some complex pattern in a string? No need for nasty hacks:

```
#include <regex>

std::string text = "Quick brown fox";
std::regex vowel_re("a|e|i|o|u");
std::regex_replace(text, vowel_re, "[$&]");
```

C++ language

If you have a question:

- google for it
- Check for the problem on stackoverflow.com
- Read the relevant parts of cppreference.com
- Ask us☺