C++ Training

Introduction and basics

Olivier Parcollet

Center for Computational Quantum Physics (CCQ)

Flatiron Institute, Simons Foundation

New York

FLATIRON
INSTITUTE
Division of Simons Foundation

Outline

Thursday Oct. 24	9:00-9:15am	Introduction : Principles
	9:15-10:00am	Basics
	10:00-10:15am	**Break**
	10:15-11:00am	Standard Library: Containers & Algorithms
	11:00-11:30am	Lambda Functions
	11:30-12:00pm	STL Algorithms II
	12:00-1:00pm	**Lunch**
	1:00-1:40pm	Editing & Static Analysis
	1:40-2:10pm	Git Basics
	2:10-2:30pm	**Break**
	2:30-3:00pm	Github, Pull requests and Code Review
	3:00-3:20pm	Managing Config Files
	3:20-3:40pm	Continuous Integration
	3:40-4:25pm	CMake and App4Triqs
	4:25-5:10pm	Testing

Friday Oct. 25	9:00-9:20am	Error handling
	9:20-9:50am	Move semantics
	9:50-10:00am	**Break**
	10:00-11:00am	Concepts
	11:00-12:00pm	TRIQS library components
	12:00-1:00pm	**Lunch**
	1:00-2:30pm	C++ HandsOn
	2:30-3:30pm	Debugging
	3:30-3:45pm	**Break**
	3:45-4:15pm	Interfacing C++ and Python
	4:15-4:30pm	Documenting
	4:30-5:00pm	Profiling & Optimizing

Targeted audience

- Upgrade C++ knowledge for e.g.
  - C programmers (use higher level abstraction).
  - Python programmers (I will show some similarity).

- I assume some familiarity with the basic (?) syntax of C++
- Do not hesitate to ask questions !

- NB : Most of the material is pure C++, some TRIQS example
- Tomorrow : a little overview of TRIQS library components.

Level of C++ expertise

1. User level
   to write code using modern C++ libraries and techniques e.g. TRIQS [this session]

2. Library writer level
   Not the purpose of this session (a few notions only).

Why C++ ?

- Why not Python, Rust, D, Swift, Julia, …. ?

- Industry standard. ISO.
- Stability but also evolution ("modern" C++).
- Performance.
  - Zero cost/overhead abstractions.
    = write more expressive code, without performance penalty.
- A language designed to build libraries.
- Largely used in HPC.

What is zero cost abstraction ?

- What is *simple* should be coded *simply*
- High level and yet fast.
- Very important for readability, long term maintenance, code review.

- Common wrong idea : compact, simple, readable code is slow.
- We want simplicity (abstraction), without any performance penalty (at zero cost).
- Generic programming is essential to achieve this.
  C++17, C++20 make it a lot easier.

Simple example

- Let us look at a tiny piece of code.
- With our matrix class
  (triqs::arrays, soon pulled out of TRIQS).

```
// using nda = triqs::arrays;

int n = 5, p = 6;     // any number …
nda::matrix a(n, p); // create a matrix of dimension n x p

// Put all elements to 0
// Version 1
a = 0;

// Version 2
for (int i = 0; i< n; ++i)
 for (int j = 0; j< p; ++j)
   a(i,j) = 0;
```

- Which one is better : version 1 or version 2 ? Why ?
- Good code expresses intent, not implementation details.

A puzzle for tomorrow

- A and B : two matrices n x n, real valued.
  A function ***trace***

- We want to write

```
double r = trace (A + B);
```

- Instead of

```
double r = 0;
for (int i = 0; i < n; ++i)
      r += A(i, i) + B(i, i);
```

- A priori, zero cost abstraction seems impossible:
  - A + B computed first, before calling `trace.`
  - Scales as N^2 while hand-written code is N

What is the cost of abstraction ?

Compilation is a transformation

- Compilers do not exactly implement what you write …
  … but a code with the same visible results

- Compilers transform code (a lot), inline functions, rewrite loops (vertorization, etc…), eliminate code with no effect.

gcc-explorer tool

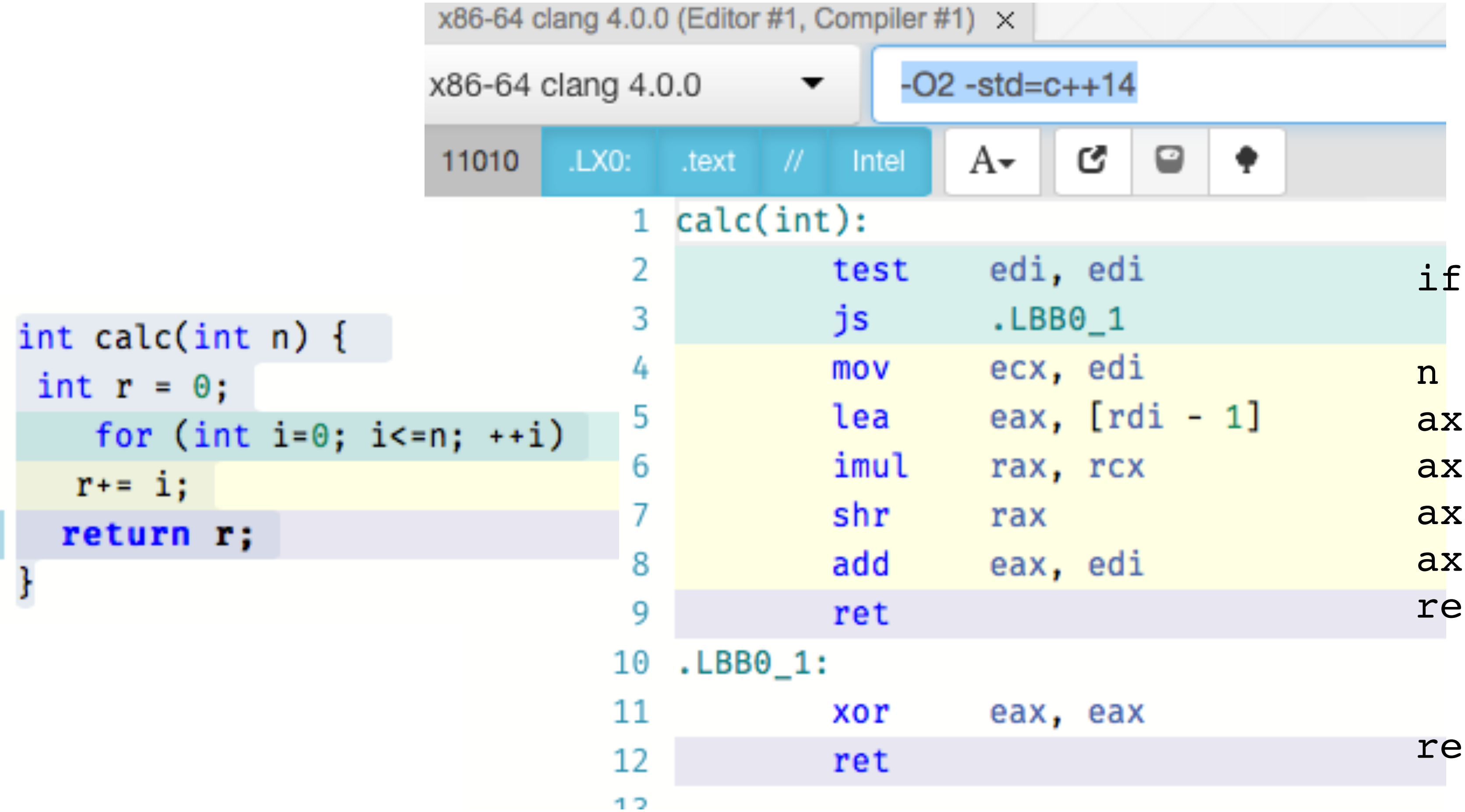
- Give a small piece of C++, it compiles, executes and shows the assembly code.
- Online : https://godbolt.org
- NB : No deep knowledge of assembly required !

Demo

Compilers can surprise you ...

- A simple function

https://godbolt.org/z/8HeLqh



x86-64 clang 4.0.0 (Editor #1, Compiler #1)

x86-64 clang 4.0.0

-O2 -std=c++14

```
int calc(int n) {
 int r = 0;
    for (int i=0; i<=n; ++i)
   r+= i;
  return r;
}
```

```
calc(int):
        test    edi, edi
        js      .LBB0_1
        mov     ecx, edi
        lea     eax, [rdi - 1]
        imul    rax, rcx
        shr     rax
        add     eax, edi
        ret
.LBB0_1:
        xor     eax, eax
        ret
```

```
if n==0 :
  goto .LBB01
n
ax = n-1
ax = n*(n-1)
ax = ax /2
ax = ax + n
return ax


return 0
```

- Compiler has replaced code by return n*(n-1)/2 + n
- Program speed sometimes hard to predict. Measure !

C++14, 17, 20 ??

C++ evolution

- A new standard now every 3 years :
  C++98, C++11, C++14, C++17, C++20, …
- Pushed by industry (e.g. Google, from mobile to large server).

- C++ is becoming simpler for users, for library writers.
- High-level constructs inspired by e.g. Python, Ocaml, …
- Standard library richer.

- Backward compatibility very rigorously enforced.
  - No change of meaning of 1/2 like in Python 2 to 3 !!

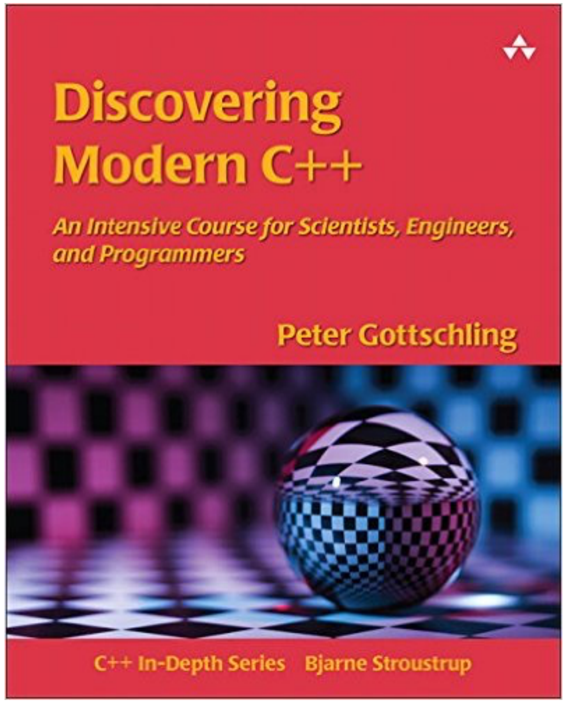
Compilers

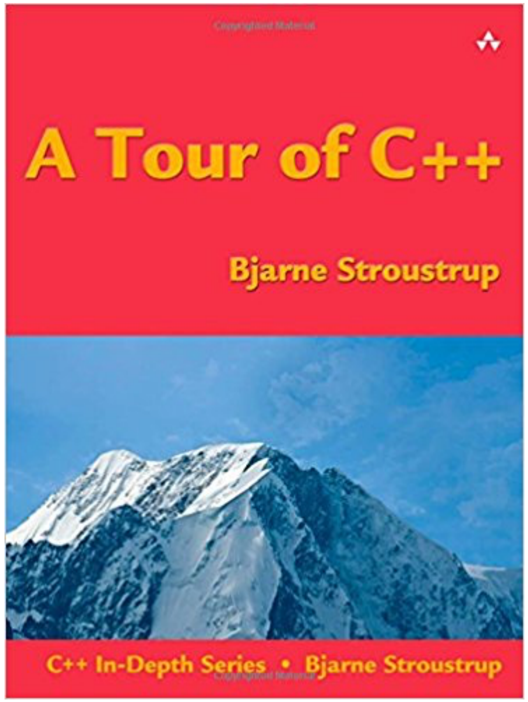
- Main compilers : gcc (GNU), llvm/clang.
- Many others, e.g. Intel.

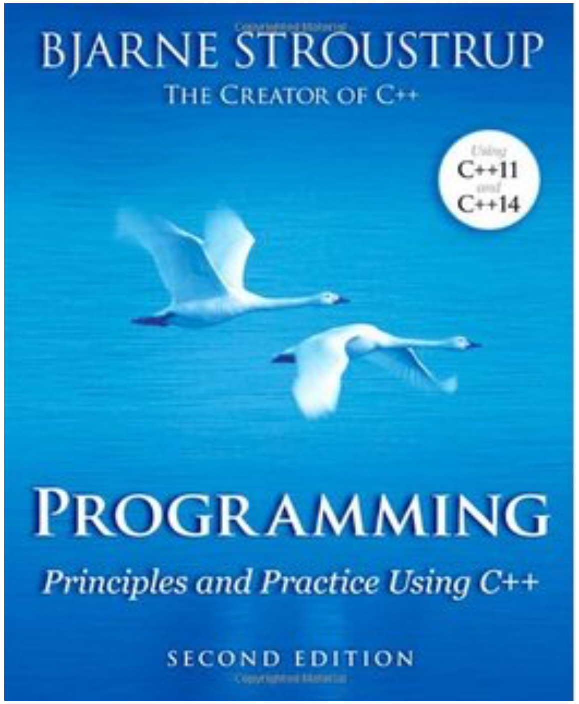
- Our policy : upgrade compilers often. Use latest versions.
  - Gcc release every year, llvm/clang every 6 months.
  - Yes ! Benefits >> Costs (our decision for TRIQS).
  - Technical solution (e.g. singularity).

Forget old books !

- How to write good C++ has changed.
- Training materials must be updated.


Discovering Modern C++
An Intensive Course for Scientists, Engineers, and Programmers
Peter Gottschling
C++ In-Depth Series Bjarne Stroustrup





Book by P. Gottschling

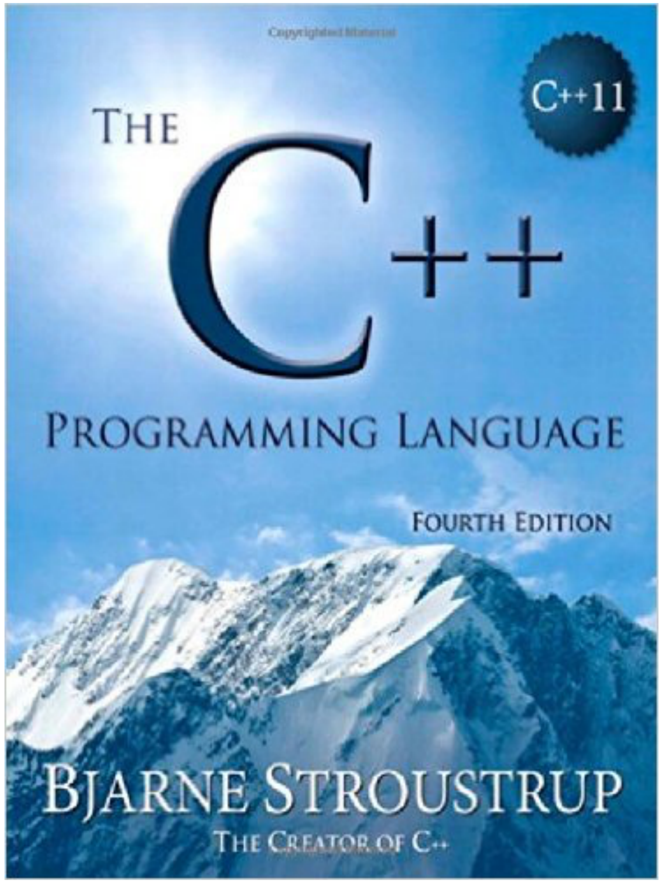
Books by B. Stroustrup

- Please, do not take random learning material on the internet !

Reference ?

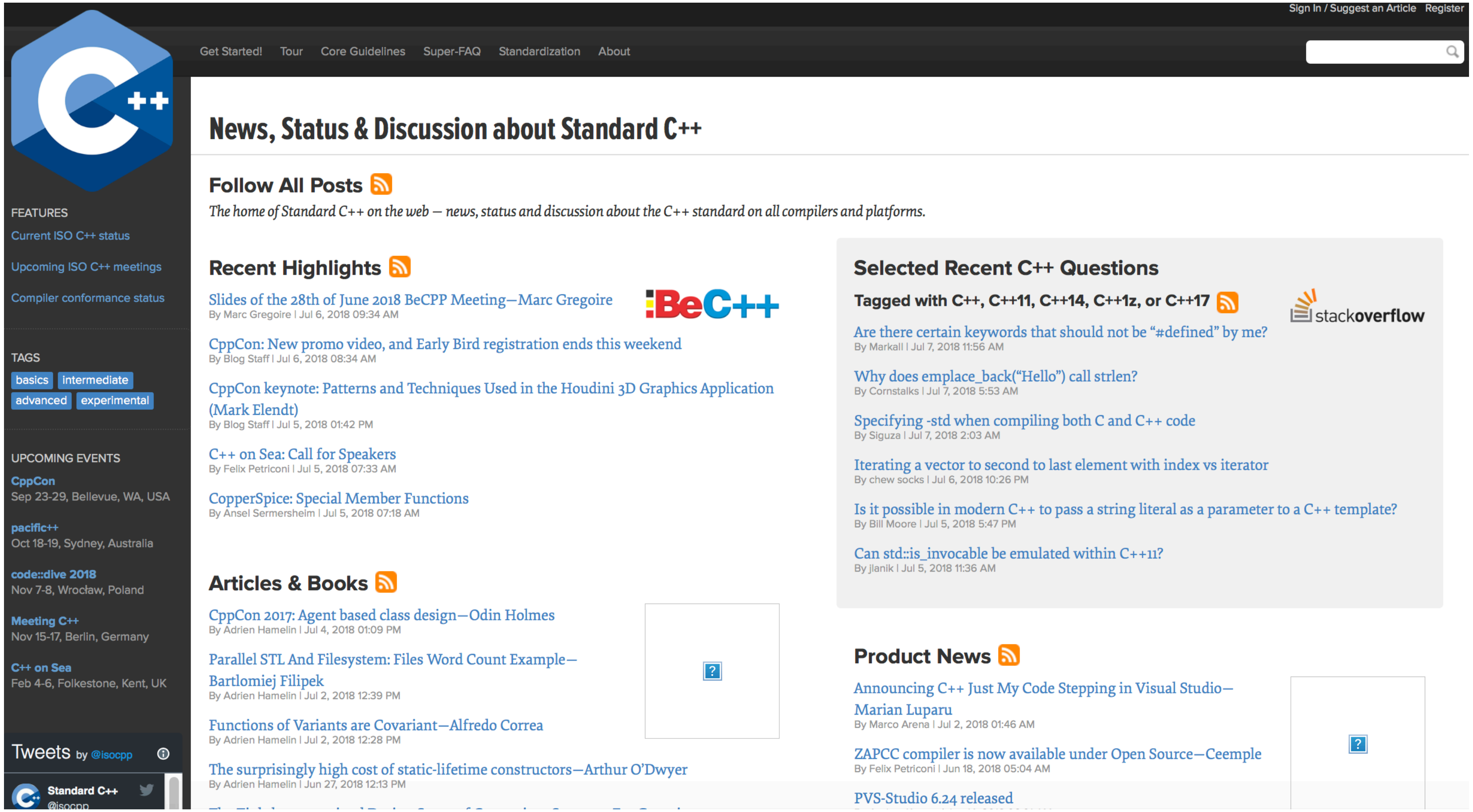
- On the web : http://en.cppreference.com/w/

- Reference book



Where to get information ?

- ISO C++ committee. E.g. check for information http://isocpp.org
- Consensus style. Backward compatibility.
- Slow, but very high quality



News, Status & Discussion about Standard C++

Follow All Posts

The home of Standard C++ on the web — news, status and discussion about the C++ standard on all compilers and platforms.

FEATURES

Current ISO C++ status

Upcoming ISO C++ meetings

Compiler conformance status

TAGS

UPCOMING EVENTS

CppCon
Sep 23-29, Bellevue, WA, USA

pacific++
Oct 18-19, Sydney, Australia

code::dive 2018
Nov 7-8, Wrocław, Poland

Meeting C++
Nov 15-17, Berlin, Germany

C++ on Sea
Feb 4-6, Folkestone, Kent, UK

Recent Highlights

Slides of the 28th of June 2018 BeCPP Meeting—Marc Gregoire
By Marc Gregoire | Jul 6, 2018 09:34 AM

CppCon: New promo video, and Early Bird registration ends this weekend
By Blog Staff | Jul 6, 2018 08:34 AM

CppCon keynote: Patterns and Techniques Used in the Houdini 3D Graphics Application (Mark Elendt)
By Blog Staff | Jul 5, 2018 01:42 PM

C++ on Sea: Call for Speakers
By Felix Petriconi | Jul 5, 2018 07:33 AM

CopperSpice: Special Member Functions
By Ansel Sermersheim | Jul 5, 2018 07:18 AM

Articles & Books

CppCon 2017: Agent based class design—Odin Holmes
By Adrien Hamelin | Jul 4, 2018 01:09 PM

Parallel STL And Filesystem: Files Word Count Example—Bartlomiej Filipek
By Adrien Hamelin | Jul 2, 2018 12:39 PM

Functions of Variants are Covariant—Alfredo Correa
By Adrien Hamelin | Jul 2, 2018 12:28 PM

The surprisingly high cost of static-lifetime constructors—Arthur O'Dwyer
By Adrien Hamelin | Jun 27, 2018 12:13 PM

Selected Recent C++ Questions

Tagged with C++, C++11, C++14, C++1z, or C++17

Are there certain keywords that should not be "#defined" by me?
By Markall | Jul 7, 2018 11:56 AM

Why does emplace_back("Hello") call strlen?
By Cornstalks | Jul 7, 2018 5:53 AM

Specifying -std when compiling both C and C++ code
By Siguza | Jul 7, 2018 2:03 AM

Iterating a vector to second to last element with index vs iterator
By chew socks | Jul 6, 2018 10:26 PM

Is it possible in modern C++ to pass a string literal as a parameter to a C++ template?
By Bill Moore | Jul 5, 2018 5:47 PM

Can std::is_invocable be emulated within C++11?
By jlanik | Jul 5, 2018 11:36 AM

Product News

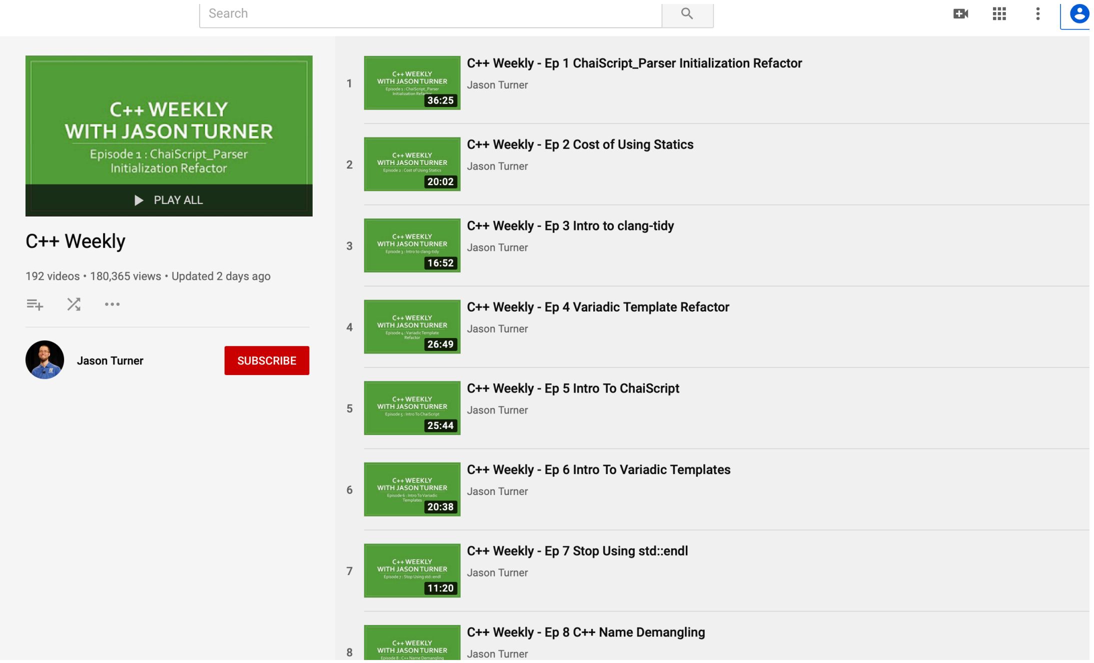
Announcing C++ Just My Code Stepping in Visual Studio—Marian Luparu
By Marco Arena | Jul 2, 2018 01:46 AM

ZAPCC compiler is now available under Open Source—Ceemple
By Felix Petriconi | Jun 18, 2018 05:04 AM

PVS-Studio 6.24 released

Where to get information ?

- Blog, e.g. “Fluent C++” https://www.fluentcpp.com
- C++ weekly on youtube https://www.youtube.com/user/lefticus1



C++ Weekly

192 videos • 180,365 views • Updated 2 days ago

Jason Turner

1 36:25 C++ Weekly - Ep 1 ChaiScript_Parser Initialization Refactor
Jason Turner

2 20:02 C++ Weekly - Ep 2 Cost of Using Statics
Jason Turner

3 16:52 C++ Weekly - Ep 3 Intro to clang-tidy
Jason Turner

4 26:49 C++ Weekly - Ep 4 Variadic Template Refactor
Jason Turner

5 25:44 C++ Weekly - Ep 5 Intro To ChaiScript
Jason Turner

6 20:38 C++ Weekly - Ep 6 Intro To Variadic Templates
Jason Turner

7 11:20 C++ Weekly - Ep 7 Stop Using std::endl
Jason Turner

8 C++ Weekly - Ep 8 C++ Name Demangling
Jason Turner

Subset of C++

Subset of C++

- C++ is a vast, multipurpose language
- “Multi-paradigm” :
  imperative functional programming,
  generic programming, object orientation
- We use a subset of the language
- Discard some old features (too verbose, unsafe)
  Instead use some recent features.
- Also some C++ (mandatory) style guidelines
- Which subset ? How to enforce it (compiler, tool) ?

Subset of C++ : highlights

- No inheritance (almost).
- Prefer the functional style :
    - Functions that do not change their arguments (purity)
    - “Data oriented”
    - Use good libraries and containers.
- Use generic programming and compile time decisions. Becoming mainstream.

Coding guidelines

- How to enforce a subset of C++ ? (like "safe" part of D e.g.).
- Rules and/or recommendations for coding style
  - e.g. C++ Core Guidelines
  - *https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md*
  - *https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-ptr*
- Team/project specific (e.g. TRIQS).

Static analysis of code

- Compiler
  - Basic, but you should use -Wall -Wextra et al.
  - (Most) compilers warnings must be resolved
- clang-tidy
  - A tool of the clang family
  - Detects violation of guidelines, some bugs, it can even fix/modernize the code.
  - Choose which guidelines to enforce in the team
  - Some are integrated in IDE, e.g. CLion, VsCode.

Basics

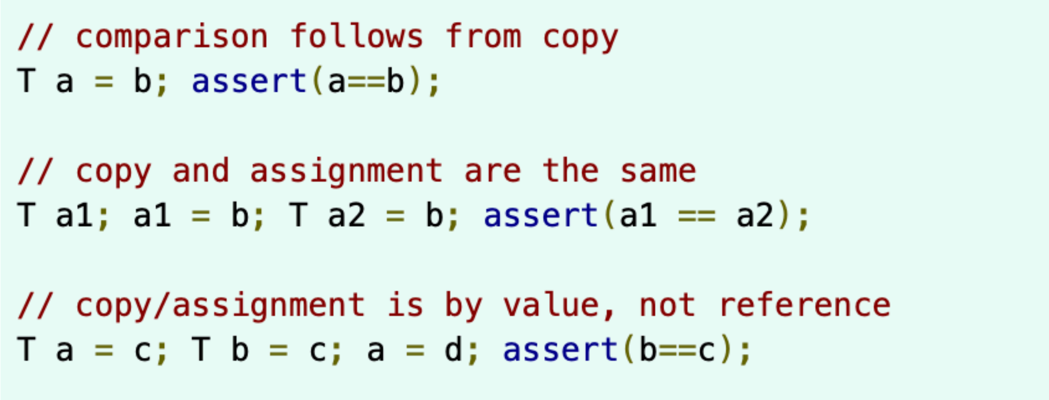
A selection of topics

Value and references

- C++ manipulates two very different kind of things.
  - Regular types.
  - References or pointers.

Regular types

- Obey a set of simple rules/axioms :
  - Copying make a new object
  - Assignment is deep.
  - Default constructible
  - …
- Simplest example : int, double.
  - *std::vector<T>*
    a simple container with N elements of type T, contiguous in memory
- Try to make your types Regular



```
// comparison follows from copy
T a = b; assert(a==b);

// copy and assignment are the same
T a1; a1 = b; T a2 = b; assert(a1 == a2);

// copy/assignment is by value, not reference
T a = c; T b = c; a = d; assert(b==c);
```

```
int a = 3;

int b1 = a;

int b2;
b2 = a;
```

```
std::vector<int> a = {1, 2, 3};

std::vector<int> b1 = a;

std::vector<int> b2;
b2 = a;
```

Regular type (2)

- Composition

```
struct A {
  int N;
  double x;
  B b; // B is regular
};
```

- A is a regular type.
- Compiler can generate good default for copy, assignment, …

Reminder : template class in C++

```
template<typename T>
class vector {

  // some array of T and many methods...

};
```

- For each T, the compiler will generate a new code

Reminder : template function in C++

```
template <typename T> T inc(T x) {
 return x + 1;
}
```

- A function for all type T
- inc can be called for an int, double,
  anything for which x +1 makes sense.
- In each case, the compiler will generate a new piece of code, optimize it, etc.
- Called "generic programming"

References : reminder

```
int i = 10;

int & r = i; // A reference to i

r = 3;   // now i AND r contains 3

i = 4; // now i AND r contains 4

int const & r = i; // a reference but I can not change its value.
```

- Another name of the same variable.
- References are :
  - Very cheap to construct. No copy of the object.
  - Immutable (one can not reassign them).

Reminder : ref usage

- Pass by reference

```
int f(A const & a) {
   // use a, no copy, no modification
}
int f(A & a) {
   // Can modify a, no copy.
}
```

- See a part of a big object

```
class A {
  std::vector<int> _data;
  int something_else;

  public:
  std::vector<int> const & data() const { return _data;}
};

// usage
A a;
do_something(a.data());
```

Reminder : const

- Something that should not be changed

```
int const a = 10;

int b = 23;
auto const & c = b;
```

```
struct A {
  int u = 10;
  int g(int v) const {
     // the method does not change u
     return u+v;
  }
};
```

```
int f(A const & a) {
  // use a, no copy, no modification
}
int f(A & a) {
  // Can modify a, no copy.
}
```

Example of non regular type

```
struct A {
  int N;
  double & x;
};
```

- Issues : copy ? Use with STL, generic algorithm ?
- Try to avoid such types, if not necessary.

const correctness

const correctness

- Make everything const by default everywhere possible.
- Const correctness is painful to fix at a later stage.

Good

```
int f (A const & a) {
 // a can not be changed.
}
```

Compiles but ***bad***
Does not pass code review

```
int f (A & a) {
 // if a is not changed.
 // compiles but bad style
}
```

Intent is : f modify x

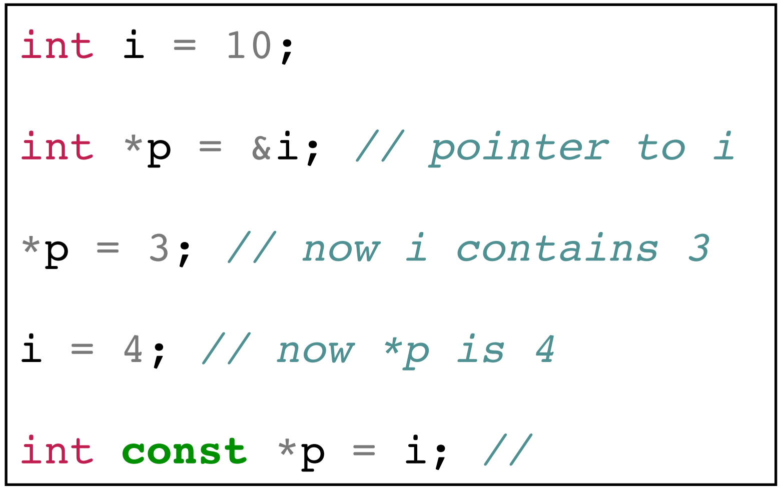
```
struct A {
  int u = 10;
  int g(int v) const { return u+v};
};
```

```
struct A {
  int u = 10;
  int g(int v) { return u+v};
};
```

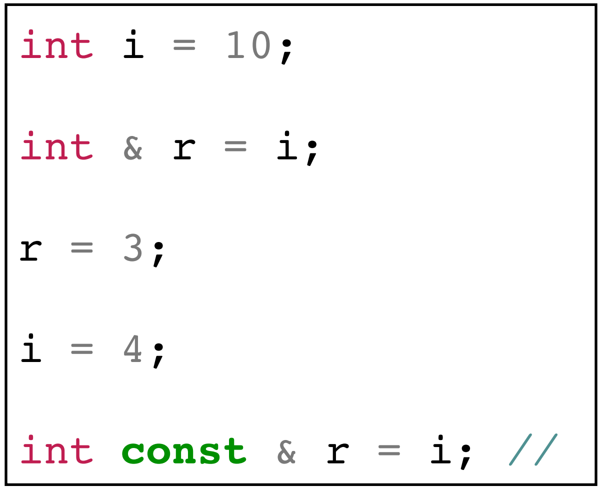
Pointers…

Pointers : reminder

- A pointer is the address of a variable
- Similar to references.
  Differences : Syntax. Can be nullptr, can be reassigned.
  Can be in an invalid state.



```
int i = 10;

int *p = &i; // pointer to i

*p = 3; // now i contains 3

i = 4; // now *p is 4

int const *p = i; //
```



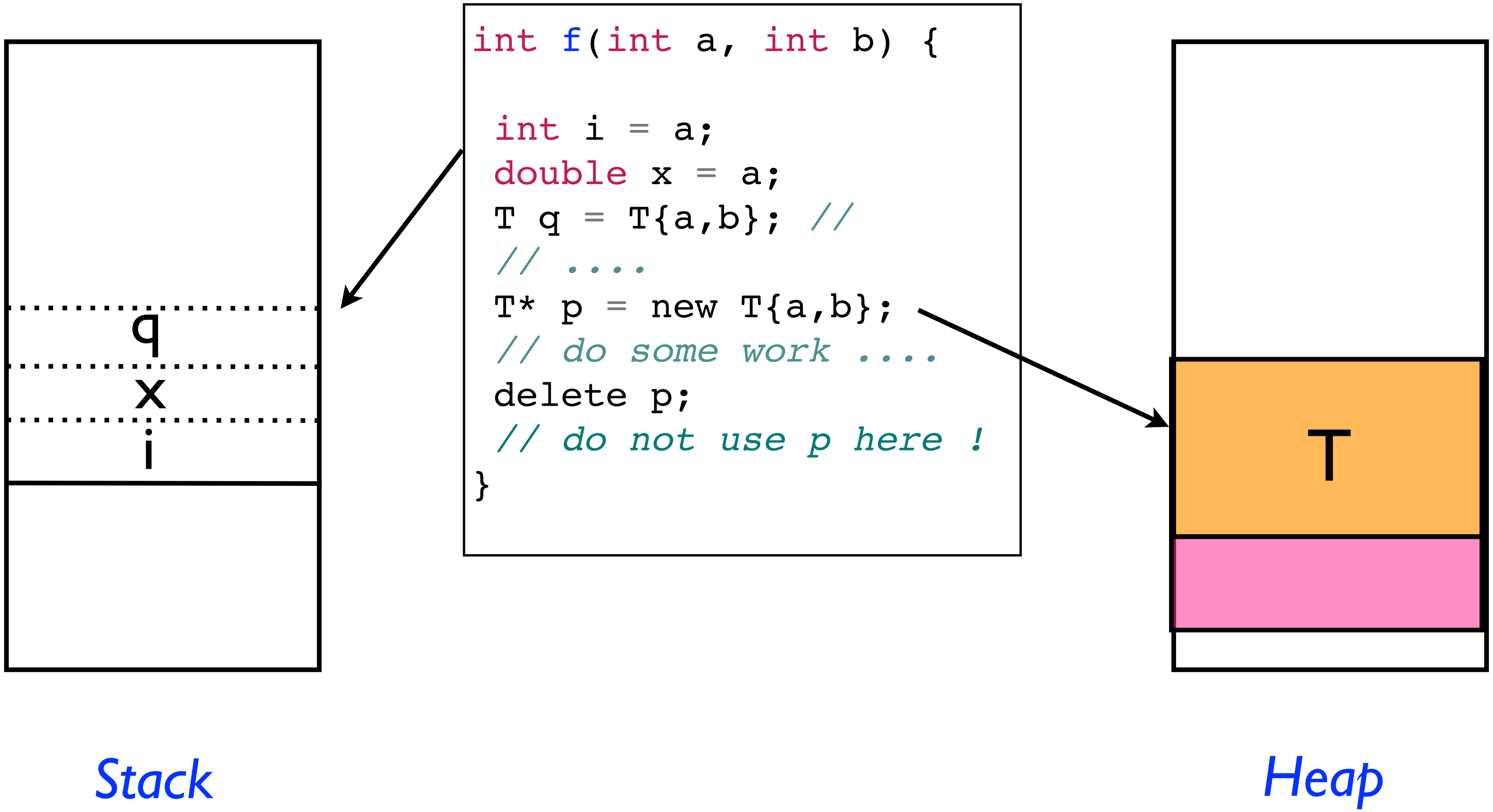
```
int i = 10;

int & r = i;

r = 3;

i = 4;

int const & r = i; //
```

Pointers in modern C++

- Pointers (or iterators) are everywhere in C / C++
- Modern C++ use much less pointers

Reminder : stack vs heap

- Stack allocation vs heap allocation



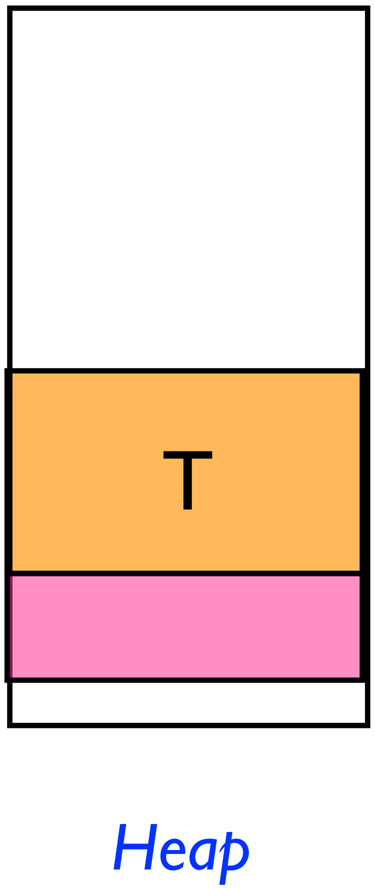
```
int f(int a, int b) {

 int i = a;
 double x = a;
 T q = T{a,b}; //
 // ....
 T* p = new T{a,b};
 // do some work ....
 delete p;
 // do not use p here !
}
```

- Local variables
- Destroyed automatically at }

- Outlives } unless explicitly deleted

Classical C/C++ issue with pointers

```
T* make_big_object_on_heap(int a, int b)
{
 T* p = new T{a,b};
 return p;
}
```



- p owns the data on the heap.
  Who is in charge of deleting p ?
- Two classical type of bugs in C/C++:
  - Memory leaks : nobody takes care of delete
  - Dangling pointers : use after delete. segfault ! ?
- Solution :
  - We can now detect these bugs easily (soon even at compile time !)
  - Use higher level abstractions which avoid these issues

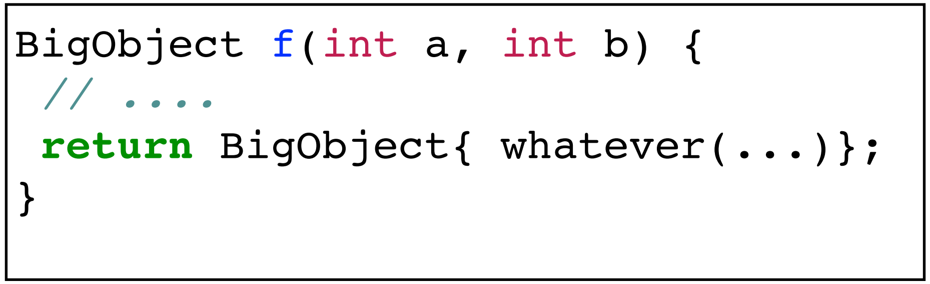
Modern C++ : Use containers

- std::vector, array/matrix class, gf class in TRIQS, …
- They handle the memory allocation for you
- They are regular types

- C++ style :
  - Do NOT use owning raw pointers
  - NB : Pointers are fine to observe an object, without ownership.

Returning a large object

- A function that builds and return a large object
- e.g. std::vector, matrix, array



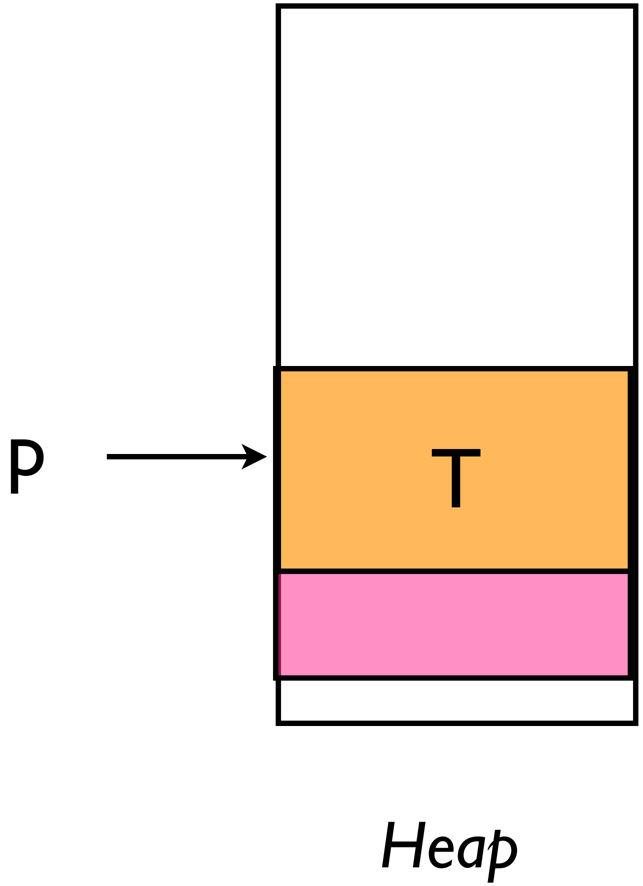
```
BigObject f(int a, int b) {
 // ....
 return BigObject{ whatever(...)};
}
```

- Simplest code (just return it) is also the most efficient
- Before C++11, there was a copy at return (even if compilers sometimes optimized it).
- Hence many books have obsolete patterns : return by pointer, etc..
- *Return Value Optimization (RVO)*. Cf also move semantics, day 2.

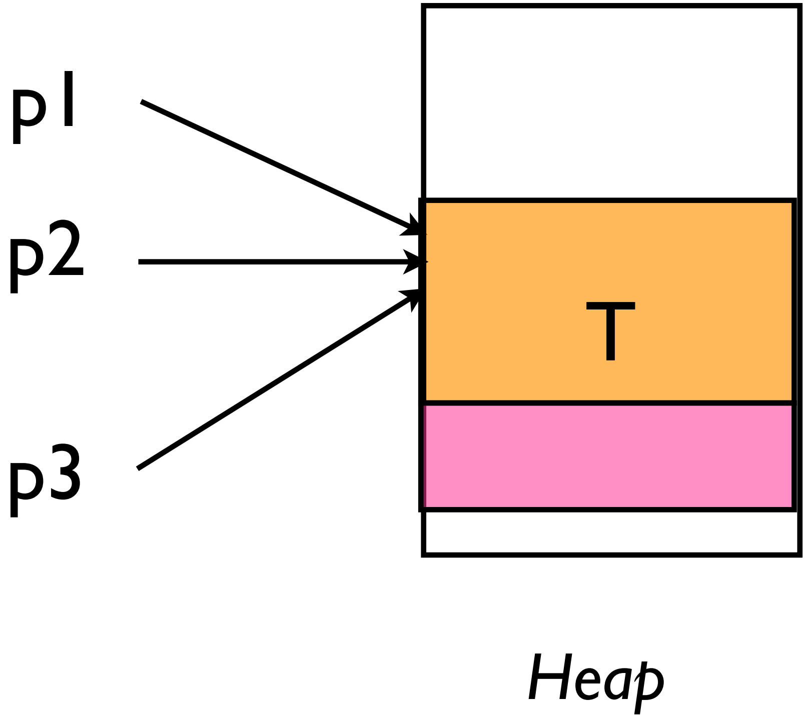
Smart pointers

- If no container is available (most likely you will not need it)

- std::unique_ptr
  - One pointer max to each object
  - p can not be copied.



- std::shared_ptr
  - Multiple pointers
  - With reference counting, like Python
  - When #ref = 0, object is destroyed



Guideline

- Use Containers (std::vector, array/matrix class, gf class…)
- Use smart pointers (unique_ptr first, shared_ptr maybe ?)
- Do NOT use owning raw pointers
    - Never use new/delete (does not pass code review !)

Scopes

}

Scopes : reminder

- Delimited by { }

```
{
  int i =9;
  std::vector<int> v= {1,2,3,4};

  // work
} // everything is cleaned up here …
```

```
// a function is a scope
int f(int n) {

}
```

```
struct A {
  A (int i);
  ~A();
};
```

- Each object has a constructor (at initialization) …
- … and a destructor (executed at })
- Guaranteed to be executed, even in case of exceptions.

RAII pattern

- A common C++ idiom : Resource Acquisition Is Initialization (?!).

```
// code defined some variable
double a;
gf<imtime, matrix_valued> { whatever  };

{
  h5::file f("myfile.h5", 'w');

  h5_write(f, a, "A");
  h5_write(f, g, "g");

} // DONE. File is closed here because f is destroyed.
```

- Similar to Python “with…”
- Releasing resource (memory, file, …) is automatic

RAII : example of unique_ptr

- std::unique_ptr
  pointer to an object, no other pointer pointing to it
- Not copy, can be moved.

```
{
 T* p = new T{a,b};
 //
 // do a lot ....

 delete p;
}
```

Bad
Should not pass code review

```
{
 std::unique_ptr<T> p = std::make_unique<T>(a,b);
 // work ...
} // delete is automatique
```

Ok

```
{
  T x{a,b};
  // work ...
} // delete is automatique
```

Even simpler

auto

auto : automatic type deduction

- Ask the compiler to deduce the type for us

```
auto y = x;       // makes a copy of x
auto z = f(a,b); // same from the result of f
```

```
auto v = std::vector<int>{1, 2, 3};
```

- NB : the type is still fixed at compile time (not dynamical as in Python)

- Enforce a type ?

```
auto x = whatever;
```

vs

```
auto x = T{whatever};

T x {whatever};
```

- Declare multiple variables. *Structure binding*

```
auto [x, y] = std::make_tuple(1, 4.3);
```

Almost always auto (AAA)

- We recommend to use auto in most place

```
auto a = A { arguments};

auto result = my_function(a);

auto m = my_matrix{ ...};
auto g = gf<imtime>{…};

int i =0;
```

- Makes code simple and more regular
- Helps a lot in generic code. Can save from unwanted conversion.
- Sometimes, e.g. for lambda, there is no other choice (Cf Nils' talk).

Loops

A simple loop

- A simple loop in Python …

```
v = [1,3,5,9]
s = 0
for x in v:
  s+=x
```

- … C++ equivalent. Main difference is types.

Intuitive

```
auto v = std::vector<int> {1,3,7,9};
int s = 0;

for (auto x : v) {
  // do something …
  s+= x;
}
```

Loops : avoid copies

```
auto v = std::vector<int> {1,3,7,9};
int s = 0;

for (auto x : v) {
  // do something !
  // may be quite complex
  s+= x;
}
```

- More generally : const, not const versions

```
auto v = std::vector<BigType> {/*...*/};

// v is unchanged, all elements are visited.
for (auto const& x : v) {
  // ...
}

// all elements visited, they can change
for (auto& x : v) {
  // ...
}
```

Simpler than what ?

- Modern C++

```
for (auto const & x : v) {
  // do something !
  s+= x;
}
```

- Intent is clearer :
  - *. Iterate on every elements in order*
  - *. v unchanged*
- As or more efficient.

- Old C++

```
for (std::vector<int>::const_iterator it=
v.begin(); it != v.end(); ++it) {
  // do something !
  s+= *it;
}
```

```
for (int i = 0; i < v.size(); ++i)
{
  // do something !
  s+= v[i];
}
```

Like Python itertools

- More sophisticated iterations.

```
#include <itertools/itertools.hpp>
std::vector<int> vec;
// ...
for (auto [n, x] : itertools::enumerate(vec)) {
   // (0, x[0]), (1, x[1]), (2, x[2]), …
}
```

```
std::vector<int> vec2, vec1;
// ...

for (auto [x, y] : itertools::zip(vec1, vec2)) {
  // (x[0], y[0]), (x[1], y[1]), (x[2], y[2]), …
}
```

- A simple header file, pulled out of TRIQS. Apache 2 licence.
- C++20 : ranges will be part of C++ std library.

https://godbolt.org/z/Yh4nR0

A quick look into C++20

- Implementing such things in C++ will become very simple …
- Coroutines
    - Generators like in Python.
    - And much more …

```python
def enumerate(X) :
    n=0
    for x in X:
        yield n, x
        n +=1
```

Python

```cpp
template<typename T>
std::generator<std::pair<n, typename T::value_type>>

enumerate(T const & x) {
  int n=0;
  for (auto const & y : x) {
    co_yield std::pair{n,y};
    n++;
  }
}
```

C++20

- NB : still use libraries by default, do NOT reimplement yourself…

Summary of basic guidelines

- Use Regular types
- Use containers. Do not use raw owning pointers
- Use const by default
- RAII : destructors clean after you.
- Use auto (AAA)
- Use expressive loops

Thank you for your attention