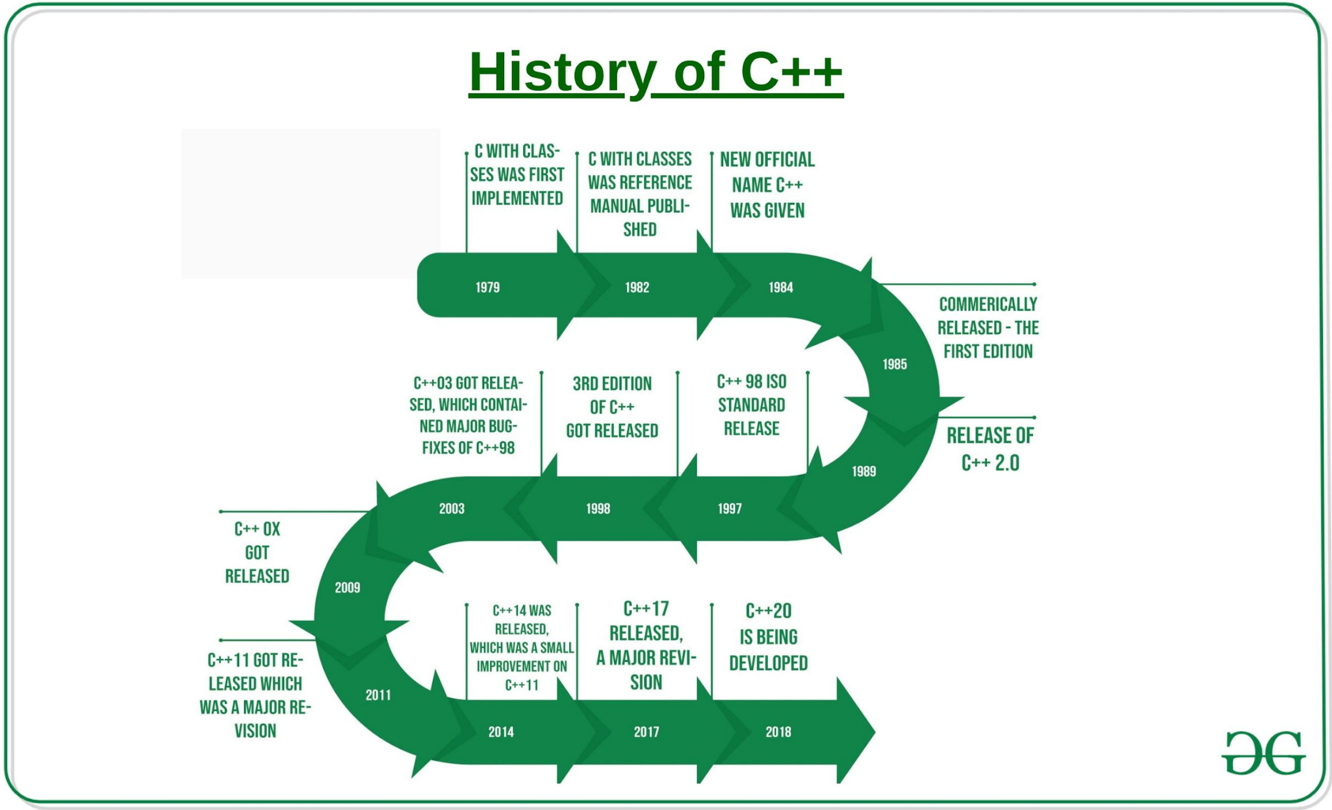
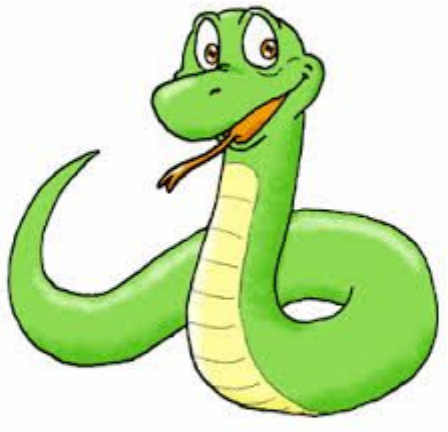
C++11 and beyond

andrea.rizzi@unipi.it

C++11/14/17/20 … C++ becoming more *pythonic*





https://github.com/AnthonyCalandra/modern-cpp-features

The “auto” keyword

- “auto” allows to deduce the type from initialization
- Details on “const”-ness or ref vs value has to be specified
  - You can also specify that you want a pointer, but it is pointless :-)

```cpp
int main(){
 auto a = 3.14; // double
 auto b = 1; // int
 auto& c = b; // int&
 auto* d = new auto(123); // int*
 auto g = new auto(123); // int*
 const auto h = 1; // const int
 auto i = 1, j = 2, k = 3; // int, int, int
 //auto l = 1, m = true, n = 1.61; // error -- `l` deduced to be int, `m` is bool
 //auto o; // error -- `o` requires initializer
}
```

Decltype,declval and decltype(auto)

- **decltype** keyword return the TYPE that would be returned by the given expression
  - declval can be used as a placeholder for an object if you know the class

```
int a = 1; // `a` is declared as type `int`
decltype(a) b = a; // `decltype(a)` is `int`
const int& c = a; // `c` is declared as type `const int&`
decltype(c) d = a; // `decltype(c)` is `const int&`
decltype(123) e = 123; // `decltype(123)` is `int`
std::vector<int> v;
std::vector<decltype(v.size())> aa; //std::vector<long unsigned int>
```

```
#include <vector>
template <class T>
std::vector<T> toVector(const T & x){
 std::vector<T> res;
 res.push_back(x);
 return res;
}

int main()
{
   decltype( toVector(std::declval<int>() ) ) a;
   //"a" is std::vector<int>
}
```

- **decltype(auto)** is like **auto** but knows **&** and **const**

```
const int x = 0;
auto x1 = x; // int
decltype(auto) x2 = x; // const int
int y = 0;
int& y1 = y;
auto y2 = y1; // int
decltype(auto) y3 = y1; // int&
```

Lambda functions

- Similar to python lambdas
  - Allow to define a function, even without a name, where you need it

```cpp
auto identity = [] (auto a) {return a;};
```

```cpp
#include <vector>
#include <algorithm>
#include <iostream>
int main()
{
   float a[] = {5.,3.1,1.0,7.32};
   std::vector<float> v(a,a+4);
   sort(v.begin(),v.end(), std::greater<float>());
   for(size_t i=0;i<v.size();i++) std::cout << v[i] << std::endl;
   sort(v.begin(),v.end(), [](auto a,auto b){return a>b;} );
   for(size_t i=0;i<v.size();i++) std::cout << v[i] << std::endl;

}
```

http://cpp.sh/2jrrh

Lambda function capture

- `[]` - captures nothing.
- `[=]` - capture local objects (local variables, parameters) in scope by value.
- `[&]` - capture local objects (local variables, parameters) in scope by reference.
- `[this]` - capture `this` pointer by value.
- `[a, &b]` - capture objects `a` by value, `b` by reference.

```cpp
int x = 1;

auto getX = [=] { return x; };
getX(); // == 1

auto addX = [=](int y) { return x + y; };
addX(1); // == 2

auto getXRef = [&]() -> int& { return x; };
getXRef(); // int& to `x`
```

- By default, value-captures cannot be modified inside the lambda because the compiler-generated method is marked as **`const`**.
- The **`mutable`** keyword allows modifying captured variables.
    - It must be specified after (....), so () is needed if no arguments are passed

```cpp
int main()
{
   int x=1;

   //does not compile
   //auto add = [x]() { x+=1;};
   //add();

   auto add = [x]() mutable { x+=3;};
   add();
   std::cout << x << std::endl;  //1
   auto addR = [&x]()  { x+=10;};
   addR();
   std::cout << x << std::endl;  //11
}
```

http://cpp.sh/956iq

Trailing return types

- Alternative way of specifying the return type of a function
  - i.e. do not get scare if you see a “->”
- Not so useful in C++14 as “auto” return type is allowed

```
int f() {
  return 123;
}
// vs.
auto f() -> int {
  return 123;
}
```

```
// NOTE: This does not compile!
template <typename T, typename U>
decltype(a + b) add(T a, U b) {
    return a + b;
}

// Trailing return types allows this:
template <typename T, typename U>
auto add(T a, U b) -> decltype(a + b) {
    return a + b;
}
```

```
int main()
{
auto identity = [](auto x) -> decltype(x) { return x; }; //C++11
auto identity = [](auto x) { return x; }; //C++14
std::cout <<  identity("foo") << " " << identity(3) << std::endl;
}
```

Initializer list, delegating constructors and converting constructors

- std::initializer_list<T> type is created when initializing variables with a list syntax like {element1, element2, ...}
- Constructors taking multiple arguments can be automatically called using initializer list
- A constructor can delegate actual construction to another constructor of the class (i.e. same syntax as what you would do with the base class)

```
#include <iostream>
struct Complex{
  Complex(float r,float i) : r(r),i(i) {}
  Complex(float r) : Complex(r,0) {}
  float r,i;
};

int main()
{
 Complex i= {0,1};
 std::cout << i.r << " " << i.i <<std::endl; // 0 1
 Complex real(7);
 std::cout << real.r << " " << real.i<<std::endl; // 7 0
}
```

http://cpp.sh/3ykoy

Range-based for loops

- Get rid of complex iterator syntax
  - `for(std::vector<int>::const_iterator it=a.begin(); it!=a.end();it++)`

```
#include <vector>
#include <iostream>
using namespace std;
int main()
{
std::vector<int> a {1, 2, 3, 4, 5};

for (auto x : a) {cout << x << endl;}

for (int& x : a) x *= 2;

for (const auto & x : a) {cout << x << endl;}
}
```

http://cpp.sh/7d7ga

override keyword

- New keyword “override” to specify that you intend the new function to override a virtual in base class
  - Throw compile time error if you are effectively not doing so
  - -Wsuggest-override warning can be generated to avoid accidentally overriding

```
struct A {
  virtual void aVirtual();
  virtual void aPureVirtual()=0;
  void aNonVirtualbar();
};

struct B : A {
  void aVirtual() override; // correct
  void aPureVirtual() override; // correct
  //void aNonVirtual() override; // error
  //void another() override; // error
};
```

constexpr

- Some expressions are just constants (i.e. they return the same value in any execution of the program as they do not depend on “inputs”)
- **constexpr** keyword allow to evaluate the result at compile time
- The code is never executed at run time, the (compile time computed) result is just placed in place of the function call

```
constexpr int square(int x) {
  return x * x;
}

int square2(int x) {
  return x * x;
}

int a = square(2);  // mov DWORD PTR [rbp-4], 4

int b = square2(2); // mov edi, 2
                    // call square2(int)
                    // mov DWORD PTR [rbp-8], eax
```

```
struct Complex {
  constexpr Complex(double r, double i) : re(r), im(i) { }
  constexpr double real() { return re; }
  constexpr double imag() { return im; }

private:
  double re;
  double im;
};

constexpr Complex I(0, 1);
```

Variadic templates

- Variadic templates allow to have functions with variable number of parameters of different types

```
#include <algorithm>
#include <iostream>
using namespace std;
template <typename First, typename... Args>
auto sum(const First first, const Args... args) -> decltype(first) {
  const auto values = {first, args...};
  return std::accumulate(values.begin(), values.end(), First{0});
}

int main(){

  cout << sum(1, 2, 3, 4, 5) << endl; // 15
  cout << sum(1) << endl;             // 1
  cout << sum(1.5, 2.0, 3.7) << endl; // 7.2
}
```

http://cpp.sh/44sjh

More complex example with variadic

- An actual example I had to write!!

```
template <typename type, typename Vec,typename... OtherVecs>
auto vector_map_t(const Vec & v,  const OtherVecs &... args) {
  ROOT::VecOps::RVec<type> res(v.size());
  for(size_t i=0;i<v.size(); i++) res[i]=type(v[i],args[i]...);
  return res;
}

template <typename func, typename Vec,typename... OtherVecs>
auto vector_map(func f, const Vec & v,  const OtherVecs &... args) {
  ROOT::VecOps::RVec<decltype(f(std::declval<typename Vec::value_type>(),
                 std::declval<typename OtherVecs::value_type>()...))> res(v.size());
  for(size_t i=0;i<v.size(); i++) res[i]=f(v[i],args[i]...);
  return res;
}
```

Smart pointers

- Smart pointers are now in the std
  - `std::unique_ptr`: no copies, if copied the original is reset
  - `std::shared_ptr`: reference counted, object is deleted when the last shared_ptr is deleted
  - `std::weak_ptr`: like shared_ptr but need to be “locked” to be accessed, when not locked can be deleted by others
- While they can be created with “new” it is best to use
  - std::make_unique and std::make_shared
    - Avoid usage of new
    - It is exception safe

http://cpp.sh/8vawp

```cpp
#include <iostream>
#include <memory>
struct Test {
    Test() { std::cout << "Created" << this << std::endl;; }
    ~Test() { std::cout << "Deleting" << this<< std::endl;; }
    void afunc() { std::cout << "Test::afunc()\n"; }
};
void f(const Test &o){     std::cout << "pointer to o: " << &o << std::endl;  }
int main()
{
    std::unique_ptr<Test> p1(new Test);  // p1 owns Foo
    if (p1) p1->afunc();
     {
        std::unique_ptr<Test> p2(std::move(p1));  // now p2 owns Foo
        f(*p1); f(*p2);
        p1 = std::move(p2);  // ownership returns to p1
        std::cout << "destroying p2...\n";
     }
     if (p1) p1->afunc();
     // Test instance is destroyed when p1 goes out of scope
}
```

http://cpp.sh/3jwto

```cpp
#include <iostream>
#include <memory>
std::weak_ptr<int> gw;
void observe()
{
    std::cout << "Ccount before lock " << gw.use_count() << ", ";
    if (auto spt = gw.lock()) { // Has to be copied into a shared_ptr before usag
    std::cout << *spt <<" (count now "  << spt.use_count() << ") \n";
    } else { std::cout << "gw is expired\n";   }
}
int main()
{
    {
      auto sp = std::make_shared<int>(42);
      gw = sp;
      observe();
    }
    observe(); //sp is now out of scope
}
```

Tuple, tie, apply

- Tuples of different types can be created (extending std::pair)
- std::get<i> is used instead of “.first” and “.second”
- Values from a tuple can be assigned to a set of variables via std::tie
- std::apply calls a function using a tuple as arguments

http://cpp.sh/3zp66

```
#include<iostream>
#include<tuple>
#include<map>
using namespace std;
int main()
{
  map<float,int> m {{1.3,2},{9.5,10}};
  for(auto & kv : m) {
    float k; int v;
    tie(k,v) = kv;
    cout << k << "-> " << v << endl;
  }
}
```

http://cpp.sh/7gjxq

```
#include<iostream>
#include<tuple>
#include<algorithm>
using namespace std;
int main()
{
    std::tuple <char, int, float> a;
    a = make_tuple('a', 10, 15.5);
    cout << "The initial values of tuple are : ";
    cout << std::get<0>(a) << " " << std::get<1>(a);
    cout << " " << std::get<2>(a) << endl;

    // Use of get() to change values of tuple
    std::get<0>(a) = 'b';
    std::get<2>(a) =  20.5;
    cout << "The modified values of tuple are : ";
    cout << std::get<0>(a) << " " << std::get<1>(a);
    cout << " " << std::get<2>(a) << endl;

    char c; int i; float f;
    std::tie(c,i,f) = a;
    cout << c << " " << i << " " << f << endl;
}
```

Structured bindings (C++17)

- Initialize multiple variables from a tuple like object (e.g. a pair, tuple, array)
- Greatly simplify access to maps (iteration on maps returns a pair that we can now use to directly initialize key and value

```cpp
#include <map>
#include <iostream>
#include <string>
using namespace std;
int main()
{
  map<string,int> m { {"red",1},{"blue",2}};
  for (auto [k, v] : m) {
    cout << k << "->" << v << std::endl;
  }
}
```

```python
#Python version
m = {"red":1, "blue":2}
for k,v in m.items():
  print(k,v)
```

Arrays, unordered containers, std::begin/end

- std::array is like C arrays (performance and memory representation) but can be copied, sorted, knows its length, etc…
- Unordered containers:
  - `unordered_set`
  - `unordered_multiset`
  - `unordered_map`
  - `Unordered_multimap`
- std::begin and std::end function can be used to get begin and end of a container
  - Also works with C arrays (that do not have member functions such as .begin()

```cpp
#include <string>
#include <iterator>
#include <iostream>
#include <algorithm>
#include <array>

int main()
{
    std::array<int, 3> a1 {1,2,3} ;
    std::array<int, 3> a2;
    a2=a1;

    std::sort(a1.rbegin(), a1.rend());

    for(auto& s: a2)  std::cout << s << ' ';
}
```

```cpp
#include <iostream>
#include <vector>
#include <iterator>
template <typename T>
int size(const T& container) {
  return std::end(container)-std::begin(container);
}

int main(){
std::vector<int> vec = {2,2,43,435,4543,534};
int arr[8] = {2,43,45,435,32,32,32,32};
std::cout << size(vec) << " " << size(arr) << std::endl;
}
```

New maps and sets operations

- New “splicing” operations on set and maps
  - set::merge merges two sets
    - Intersection (i.e. elements not inserted) remains in the merged one
    - Does not “copy”
- extract/insert allow to move or replace map entries

```
std::set<int> src {1, 3, 5};
std::set<int> dst {2, 4, 5};
dst.merge(src);
// src == { 5 }
// dst == { 1, 2, 3, 4, 5 }
```

```
std::map<int, string> m {{1, "one"}, {2, "two"}, {3, "three"}};
auto e = m.extract(2);
e.key() = 4;
m.insert(std::move(e));
// m == { { 1, "one" }, { 3, "three" }, { 4, "two" } }
```

```
std::map<int, string> src {{1, "one"}, {2, "two"}, {3, "buckle my shoe"}};
std::map<int, string> dst {{3, "three"}};
dst.insert(src.extract(src.find(1))); // Cheap remove and insert of { 1, "one" } from `src` to `dst`.
dst.insert(src.extract(2)); // Cheap remove and insert of { 2, "two" } from `src` to `dst`.
// dst == { { 1, "one" }, { 2, "two" }, { 3, "three" } };
```

std::any (C++17)

- std::any (or boost::any before C++17) allows to store any type

```
#include <any>
#include <iostream>
#include <vector>
#include <string>
int main()
{

    // any type
    std::any a = 1;
    std::cout << a.type().name() << ": " << std::any_cast<int>(a) << '\n';
    a = 3.14;
    std::cout << a.type().name() << ": " << std::any_cast<double>(a) << '\n';
    a = true;
    std::cout << a.type().name() << ": " << std::any_cast<bool>(a) << '\n';

    a.reset();
    if (!a.has_value())  {  std::cout << "no value\n";   }

    std::vector<std::any> v={1.2,"hello",7,true};
    for(auto a : v) {std::cout << a.type().name() << "\n";}
}
```

```
# ./a.out
i: 1
d: 3.14
b: 1
no value
d
PKc
i
b
```

```
# ./a.out  | c++filt -t
int: 1
double: .14
bool: 1
no value
double
char const*
int
bool
```

More...

- Template variables
- Nullptr
- std::regex
- Std::optional // get rid of std::pair<what I want to retur, bool>
- std::variant

Multi threading

- Functions built-in in the language to handle multi-threading:
  - std::thread
  - std::async (can also be **lazy**)

```cpp
// thread example
#include <iostream>       // std::cout
#include <thread>         // std::thread

void foo()
{
  // do stuff...
}

void bar(int x)
{
  // do stuff...
}

int main()
{
  std::thread first (foo);     // spawn new thread that calls foo()
  std::thread second (bar,0);  // spawn new thread that calls bar(0)

  std::cout << "main, foo and bar now execute concurrently...\n";

  // synchronize threads:
  first.join();                // pauses until first finishes
  second.join();               // pauses until second finishes

  std::cout << "foo and bar completed.\n";

  return 0;
}
```

```cpp
#include <future>
#include <iostream>
#include <vector>

int twice(int m) {
  std::cout << "now computing" << m  << std::endl;
  return 2 * m;
}
int main() {
  std::vector<std::future<int>> futures;

   for(int i = 0; i < 10; ++i) {
     futures.push_back (std::async(std::launch::async,twice, i));
   }

   //retrive and print the value stored in the future
   for(auto &e : futures) {
     std::cout << e.get() << std::endl;
   }

   return 0;
}
```

std::chrono

A C++ interface to timing information

```
#include <iostream>
#include <chrono>
#include <ctime>

long fibonacci(unsigned n)
{
    if (n < 2) return n;
    return fibonacci(n-1) + fibonacci(n-2);
}

int main()
{
    auto start = std::chrono::system_clock::now();
    std::cout << "f(42) = " << fibonacci(42) << '\n';
    auto end = std::chrono::system_clock::now();

    std::chrono::duration<double> elapsed_seconds = end-start;
    std::time_t end_time = std::chrono::system_clock::to_time_t(end);

    std::cout << "finished computation at " << std::ctime(&end_time)
              << "elapsed time: " << elapsed_seconds.count() << "s\n";
}
```

Output

```
f(42) = 267914296
finished computation at Mon Oct  2 00:59:08 2017
elapsed time: 1.88232s
```

User defined literals

- You can define new literals!   (i.e. the “f” in “3.4f” that makes it a float)
    - And some are already defined by the STL (e.g. for std::chrono and strings)
- User defined literals should start with “_”

http://cpp.sh/5w4cjv

```
#include <iostream>
#include <string>
using namespace std::string_literals;

struct Complex{
  Complex(float r,float i) : r(r), i(i) {}
  float r,i;

};
Complex operator+(const Complex &a,const Complex &b) {return Complex(a.r+b.r,a.i+b.i);}
Complex operator+(float a,const Complex &b) {return Complex(a+b.r,b.i);}
Complex operator+(const Complex &b,float a) {return Complex(a+b.r,b.i);}

Complex operator "" _i(long double i) {return Complex(0,i);}

int main(){
    auto c= 7 + 4.3_i;
    std::cout <<  c.r << " " << c.i << std::endl;
    std::cout << ("hello "s + "world"s) << std::endl;
}
```

l-value and r-values

- L-value : an expression that can be on the left or on the right side of an assignment (=) operator
- R-value : an expression that can be only on the right side

… or …

- An L-value is an expression that refers to a memory location and allows us to take the address of that memory location via the & operator.
- An R-value is an expression that is not an L-value.

```
int main()
{
    int a,b;   // "a" and "b" are both l-values
    a = 8;   // "8" is a r-value
//    8 = a;   //  error: lvalue required as left operand of assignment
    b = a;   // l-value can be both on the left and on the right
    a = a+b;  // "a+b" is an r-value
//    a+b = b;  //  error: lvalue required as left operand of assignment

}
```

Move semantics

- C++ code is often full of “copies” because temporary “r-values” are created and then thrown away
  - It was not a big deal in “C” as custom types (classes) with complex (aka memory expensive) behaviour did not exist
- Temporary objects (r-values) could be sometimes simply “moved” in the the variable we want to assign them to
  - Avoid a copy
- ***r-value references*** (e.g. int &&) and std::move allow to do so
  - Much faster code avoiding copying & allocating
- std::swap can also be used, e.g. to avoid copying STL containers
- A nice example to explaining rvalues / lvalues / move semantics
  - https://www.internalpointers.com/post/c-rvalue-references-and-move-semantics-beginners
  - Let’s go through it !

A random list of C++20 new things

- Spaceship operator “ <=>"
- Ranges
- Concepts: specify the requirements a template type should have
- Designated initializers
- … many more...

```cpp
#include <vector>
#include <ranges>
#include <iostream>

int main()
{
    std::vector<int> ints{0,1,2,3,4,5};
    auto even = [](int i){ return 0 == i % 2; };
    auto square = [](int i) { return i * i; };

    for (int i : ints | std::views::filter(even) | std::views::transform(square)) {
        std::cout << i << ' ';
    }
}
```

```cpp
#include <iostream>

struct Point2D{
    int x;
    int y;
};

class Point3D{
public:
    int x;
    int y;
    int z;
};

int main(){

    std::cout << std::endl;

    Point2D point2D{.x = 1, .y = 2};          // (1)
    Point3D point3D{.x = 1, .y = 2, .z = 3};  // (2)

    std::cout << "point2D: " << point2D.x << " " << point2D.y << std::endl;
    std::cout << "point3D: " << point3D.x << " " << point3D.y << " " << point3D.z << std::endl;

    std::cout << std::endl;

}
```