Modern C++

From Pointers to Values

Francesco Giacomini – INFN-CNAF

CNAF, 2014-11-18

Outline

- An evolving language
  - C++ Standard timeline
  - *C++11 feels like a new language*
- Why pointers?
  - Typical uses
  - Issues with pointers
- Smart pointers
  - How they work
  - How to use them
- Move semantics
- Error management
- Putting it all together

Practical information

- Use a recent compiler
  - gcc 4.8+, clang 3.3+
  - compile with `-std=c++11` or `-std=c++1y` (or `-std=c++1z`)
- To check generated assembler with different compilers

  http://gcc.godbolt.org
- To compile and run with different compilers

  http://coliru.stacked-crooked.com/
- C++ reference

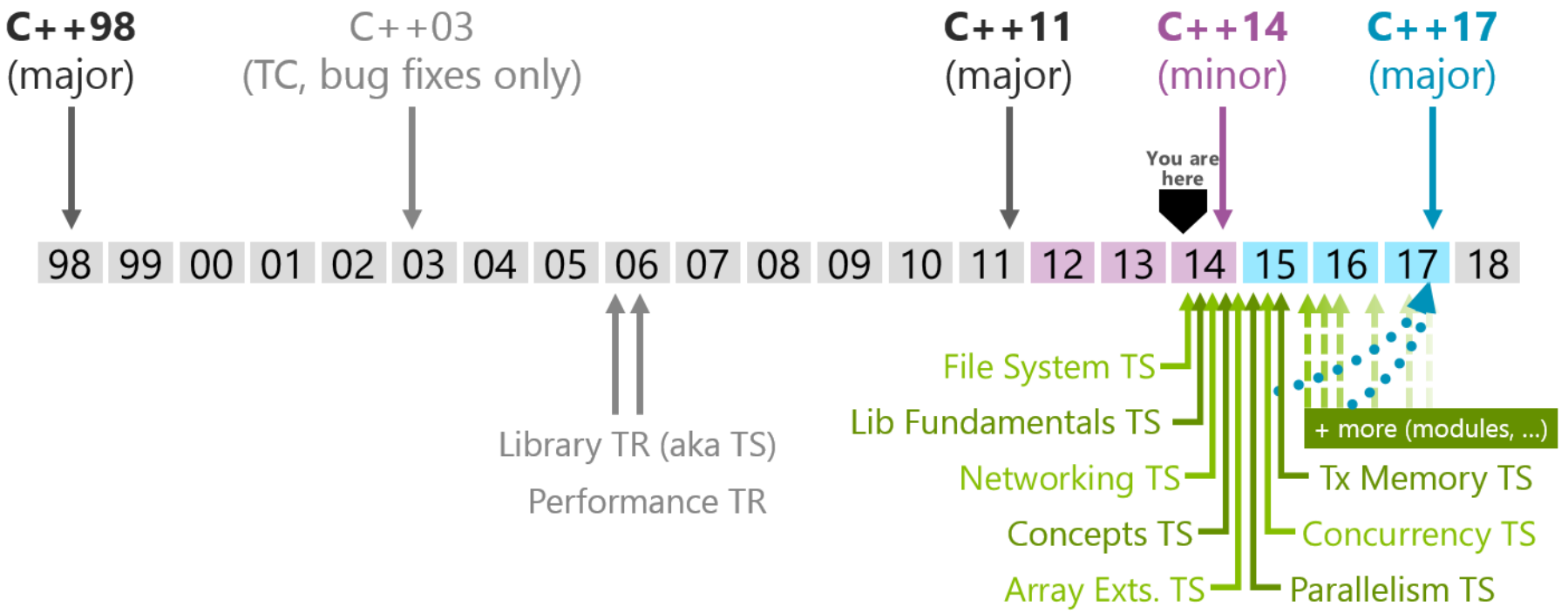
  http://en.cppreference.com/

Outline

- An evolving language
  - C++ Standard timeline
  - *C++11 feels like a new language*
- Why pointers?
  - Typical uses
  - Issues with pointers
- Smart pointers
  - How they work
  - How to use them
- Move semantics
- Error management
- Putting it all together

C++ Standard timeline

Invented in 1979 (“C with classes”), standardized in 1998



Outline

- An evolving language
  - C++ Standard timeline
  - *C++11 feels like a new language*
- Why pointers?
  - Typical uses
  - Issues with pointers
- Smart pointers
  - How they work
  - How to use them
- Move semantics
- Error management
- Putting it all together

auto

In C++ 98 we are used to write:

```
std::map<std::string, int> m;
std::map<std::string, int>::iterator iter = begin(m);
```

But, why should we tell the compiler what the type of it is? it must know!

```
std::map<std::string, int> m;
auto iter = begin(m);
```

The auto type specifier signifies that the type of a variable being declared shall be deduced from its initializer

```
auto a;                 // error, no initializer
auto i = 0;             // i has type int
auto d = 0.;            // d has type double
auto f = 0.f;           // f has type float
auto c = "hello";       // c has type char const*
auto p = new auto(1); // p has type int*
```

auto

- auto is never deduced to be a reference. If needed, & must be added explicitly

```
auto g = d;         // g has type double (g is a copy of d)
auto& h = e;        // h has type double& (h and e are aliases)
auto k = h;         // k has type double (not double&)
auto const& n = e; // n has type double const&
auto const l = h;  // l has type int const
auto& m = l;        // m has type int const&
```

- Trick to inspect the deduced type

```
template<typename T> struct TD;

TD<decltype(m)>; // the compiler error will show the type
```

Initializer lists

In C++98 initializing data structures is often burdensome

```
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(4);
v.push_back(5);

std::map<int, std::string> ids;
ids.insert(std::make_pair(23, "Andrea"));
ids.insert(std::make_pair(49, "Camilla"));
ids.insert(std::make_pair(96, "Ugo"));
ids.insert(std::make_pair(72, "Elsa"));
```

In C++11 the operation is much simpler

```
std::vector<int> const v = {1, 2, 3, 4, 5};

std::map<int, std::string> const ids = {
  {23, "Andrea"},
  {49, "Camilla"},
  {96, "Ugo"},
  {72, "Elsa"}
};
```

Favor const-correctness

→ thread-safety

Uniform initialization syntax

- Use braces {} for all kinds of initializations
  - initializer lists

```
vector<int> v {1, 2, 3};        // calls vector<int>::vector(initializer_list<int>)
vector<int> v = {1, 2, 3};      // idem
```

  - normal constructors

```
complex<double> c(1.0, 2.0);    // calls complex<double>::complex(double, double);
complex<double> c {1.0, 2.0};   // idem
complex<double> c = {1.0, 2.0}; // idem
```

  - aggregates (available in C++98 as well)

```
struct X {
  int i_, j_;
};
X x {1, 2};   // x.i_ == 1, x.j_ == 2
X x = {1, 2}; // idem
```

Uniform initialization syntax

Additional advantage: prevention of narrowing

```
int c(1.5);  // ok, but truncation, c == 1
int c = 1.5; // idem
int c{1.5};  // error
int c{1.};   // error, floating → integer is always considered narrowing

char c(7);   // ok, c == 7
char c(256); // ok, but c == 0 (assuming a char has 8 bits)
char c{256}; // error, the bit representation of 256 doesn't fit in a char
```

Beware: initializer-list constructors are favored over other constructors

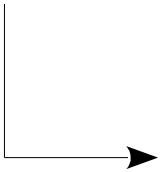
```
vector<int> v{2}; // calls vector<int>::vector(initializer_list<int>)
                  // v.size() == 1, v[0] == 2
vector<int> v(2); // calls vector<int>::vector(size_t)
                  // v.size() == 2, v[0] == v[1] == 0
```

range for

```
std::vector<Particle> v { … };

for (std::vector<Particle>::const_iterator b = begin(v), e = end(v); b != e; ++b) {
  print(*b);
}

for (std::vector<Particle>::iterator b = begin(v), e = end(v); b != e; ++b) {
  update(*b);
}
```



C++11: simplified syntax to iterate on a sequence

```
std::vector<Particle> v { … };

for (Particle p: v) { // or, rather, Particle const&
  print(p);
}
for (auto p: v) { // or, rather, auto const&
  print(p);
}
for (auto& p: v) {
  update(p);
}
for (p: v) { … } // C++17 (probably)
```

range for

- A range can be:
  - an array
    - e.g. int a[10];
  - a class C (typically a container) such that C::begin() and C::end() exist
    - e.g. std::vector<int>, std::string
  - a class C such that begin(C) and end(C) exist
- Instead of an explicit for loop consider the use of an algorithm (such as for_each or find_if), possibly with a lambda function (see later)

```
for_each(begin(v), end(v), print);

for_each(begin(v), end(v), update);
```

What is a function object?

- A function object (aka functor) is an instance of a class that has overloaded `operator()`
- A function object can then be used as if it were a function
- A function object, being an instance of a class, can have state

```
struct Incrementer
{
  int operator()(int i) const
  { return ++i; }
};

Incrementer inc;
auto r = inc(3); // int r == 4
```

```
class Add_n {
  int n_;
public:
  explicit Add_n(int n): n_(n) {}
  int operator()(int i) const
  { return n_ + i; }
};

Add_n s{5};
auto r = s(3); // int r == 8
```

[]{}

- Lamba functions
- A concise way to create simple anonymous function objects
- Useful to pass actions/callbacks to algorithms, threads, frameworks, …

```
vector<int> v = {-2, -3, 4, 1};
sort(v.begin(), v.end()); // default sort, v == {-3, -2, 1, 4}

struct abs_compare
{
  bool operator()(int l, int r) const { return abs(l) < abs(r); }
};
sort(v.begin(), v.end(), abs_compare()); // C++98, v == {1, -2, -3, 4}

sort(v.begin(), v.end(), [](int l, int r) -> bool { return abs(l) < abs(r); }); // C++11
```

optional

Lambda functions

- The evaluation of a lambda expression produces a closure, which consists of:
  - the code of the body of the lambda
  - the environment in which the lambda is defined
    - the variables that are referenced in the body need to be captured and are stored in the generated function object

```
double min_salary = …;
find_if(
  begin(employees),
  end(employees),
  [=](Employee const& e) { return e.salary() < min_salary; }
);
```

```
[]      // capture nothing
[&]     // capture all by reference
[=]     // capture all by value
[=, &i] // capture all by value, but i by reference
```

Example

- Fill a vector with N random integers between 1 and M

```
std::default_random_engine engine;
std::uniform_int_distribution<int> uniform_dist(1, M);

std::vector<int> v;
v.reserve(N);                              // sets the vector capacity
std::generate_n(
  std::back_inserter(v),                   // calls v.push_back()
  N,
  [&]() { return uniform_dist(engine); } // function that generates a number
);
```

Example

```
// sum all the elements
auto sum = accumulate(begin(v), end(v), 0);

// find the first number greater than 32
auto it = find_if(begin(v), end(v), [](int j) { return j > 32; });
assert(it == end(v) || *it > 32);

// multiply all the elements by 2
transform(begin(v), end(v), begin(v), [](int j) { return 2 * j; });

// sort in ascending order
sort(begin(v), end(v));

// sort in descending order
sort(begin(v), end(v), [](int i, int j) { return j < i; });

// Erase from the vector the multiples of 3 or 7
auto it = std::remove_if(begin(v), end(v), [](int i) { return i % 3 || i % 7; });
v.erase(it, end(v));
```

Outline

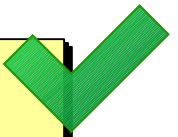
- An evolving language
  - C++ Standard timeline
  - *C++11 feels like a new language*
- Why pointers?
  - Typical uses
  - Issues with pointers
- Smart pointers
  - How they work
  - How to use them
- Move semantics
- Error management
- Putting it all together

Deal with large objects

- Function parameter

```
void modify(LargeObject* o) {
  o->f();
}
void use(LargeObject const* o) {
  o->f(); // f must be const
}
```

```
void modify(LargeObject& o) {
  o.f();
}
void use(LargeObject const& o) {
  o.f(); // f must be const
}
```



(unless the parameter is optional)

- Function return value

```
LargeObject* make_large_object() {
  return new LargeObject;
}
```

Dynamic polymorphism

- Dynamic polymorphism requires access through a pointer (or reference)

```
struct Base {
  virtual ~Base() = default;
  virtual void f() const {}
};

struct Derived: public Base {
  ~Derived() = default;
  void f() const override {}
};

Base* b = new Derived;
b->f();   // call Derived::f()
delete b; // call Derived::~Derived()
```

- Cannot pass objects because of slicing

De-virtualization

- Consider static polymorphism
  - no virtual functions → no need to de-virtualize

```
struct B {
  virtual int f() = 0;
};
struct D1: B {
  int f() override { return 42; }
};
struct D2: B {
  int f() { return 31; }
};


int f(B* b) { return b->f(); }

f(new D1);
f(new D2);
```

```
struct D1 {
  int f() { return 42; }
};
struct D2 {
  int f() { return 31; }
};

template<typename B>
int f(B* b) { return b->f(); }

f(new D1);  // calls f<D1>(D1* b)
f(new D2);  // calls f<D2>(D2* b)
```

- Next step is to pass objects (or references thereof)

pimpl idiom

- Pointer to implementation

```
// in state_machine.hh

class StateMachine {
  class Impl;
  Impl* impl_;
public:
  StateMachine(…);
  ~StateMachine();
  void configure();
  void start();
  void stop();
  // …
};
```

```
// in state_machine.cc

class StateMachine::Impl {
  // actual data members
  Impl( … ): … {}
}

StateMachine::StateMachine(…)
  : impl_{new Impl( … )}
{ … }

~StateMachine::StateMachine(…) {
  delete impl_;
}

void StateMachine::configure() {
  impl_-> … ;
}
// …
```

- Compilation firewall
  - a change in the private data members does not require a recompilation of the clients of E

Keep a link to an object

```
template<typename Container>
class back_insert_iterator: public …
{
  Container* container;
 public:
  explicit back_insert_iterator(Container& x): container(&x) { }
  // …
};

back_insert_iterator it1(v);
auto it2 = it1;                 // copy constructor
it2 = it1;                      // copy assignment
```

- Pointers are copyable, references are not
  - the copy assignment wouldn't work with

    `Container& container;`
- The link is non-owning, it's just to “visit” the pointee
  - this is fine

Direct memory manipulation

- Implementation of high-level data structures (vector, string, etc.)
  - owning-pointer

```
class string
{
  char* data_;
  size_t size_;
  size_t capacity_;
 public:
  string(…): data_(new char[…]) { … }
  ~string() { delete [] data_; }
};
```

- Access to hardware registers

```
static const unsigned int address = 0xfffe0000;

*reinterpret_cast<volatile uint8_t*>(address) = 42;
```

Interface to legacy code

- Typically, but not necessarily, C

```
DIR *opendir(const char *name);
int closedir(DIR *dirp);

auto dir = opendir("/home/giaco/my_file.txt");
closedir(dir);
```

```
void *malloc(size_t size);
void free(void *ptr);

auto p = static_cast<int*>(malloc(sizeof(int))); // void* → T* not implicit
free(p);                                         // T* → void* implicit
```

- Often functions come in pairs
  - acquire a resource (a file, a connection, a lock, ...)
  - release a resource

Arrays

- *“Arrays decay to pointers at the slightest provocation”*

```
void f(int* a);

int a1[N];             // N is const, known at compile time
int* a2 = new int[n]; // n is known only at runtime

f(a1); // calls f(int*)
f(a2); // calls f(int*)

delete [] a2;          // don't forget the delete, don't forget the []
```

- The size of the array is not encoded in the type
- Safer alternatives exist, notably `std::vector<>` and `std::array<>`

```
std::array<int, N> a1;  // N is const, known at compile time
std::vector<int> a2(n); // n is known only at runtime
```

Laziness

- “I didn't think about it”, “I started from an existing piece of code”, “I started from an example”

```
void X::f() {
  ofstream* dump = new ofstream(dump_fname_);
  dump->write(…);
  dump->close();
  delete dump;
  dump=0;
}
```



```
void X::f() {
  ofstream dump{dump_fname_};
  dump.write(…);
}
```

What's wrong with pointers?

- Am I the owner of the pointee (the object pointed to)? who is responsible for the delete?

```
char* p = strstr("giacomini", "min");
```

  - Should I free p?
  - Is p a null-terminated string? is it an array? of what size?
  - The answers are not encoded in the type

What's wrong with pointers?

- Am I the owner of the pointee (the object pointed to)? who is responsible for the delete?
- Run-time overhead (allocation, indirection, fragmentation, etc.)

What's wrong with pointers?

- Am I the owner of the pointee (the object pointed to)? who is responsible for the delete?
- Run-time overhead (allocation, indirection, fragmentation, etc.)
- Explicit allocation and deallocation increases the risk of memory leaks and double deletes

```
{
  ofstream* dump = new ofstream(dump_fname);
  // …
} // ops, forgot to delete dump

{
  ofstream* dump = new ofstream(dump_fname);
  // …
  delete dump;
  // …
  delete dump; // ops, delete again
}
```

What's wrong with pointers?

- Am I the owner of the pointee (the object pointed to)? who is responsible for the delete?
- Run-time overhead (allocation, indirection, fragmentation, etc.)
- Explicit allocation and deallocation increases the risk of memory leaks and double deletes
- Unsafe in presence of exceptions

```
{
  ofstream* dump = new ofstream(dump_fname);
  // potentially-throwing code
  delete dump; // not executed in case an exception is thrown
}
```

What's wrong with pointers?

- Am I the owner of the pointee (the object pointed to)? who is responsible for the delete?
- Run-time overhead (allocation, indirection, fragmentation, etc.)
- Explicit allocation and deallocation increases the risk of memory leaks and double deletes
- Unsafe in presence of exceptions

- In general what is said for memory is often applicable to any resource

Outline

- An evolving language
  - C++ Standard timeline
  - *C++11 feels like a new language*
- Why pointers?
  - Typical uses
  - Issues with pointers
- Smart pointers
  - How they work
  - How to use them
- Move semantics
- Error management
- Putting it all together

Smart pointers

- Objects that behave like pointers, but also manage the lifetime of the pointee
- Leverage the RAII idiom
  - Resource Acquisition Is Initialization
    - Resource (e.g. memory) is acquired in the constructor
    - Resource (e.g. memory) is released in the destructor
- Importance of how the destructor is designed in C++
  - deterministic: guaranteed execution at the end of the scope
  - order of execution opposite to order of construction

Smart pointers

```
template<typename Pointee>
class SmartPointer
{
  Pointee* p_;
 public:
  SmartPointer(Pointee* p): p_(p) {}
  ~SmartPointer() { delete p_; }
};

int main()
{
  SmartPointer<int> sp{new int};
  // …
} // automatic and guaranteed
  // destruction of sp here,
  // which calls delete p;
```

- Clear management responsibility
- Likely no space nor runtime overhead wrt to raw pointer
  - depends on the type of smart pointer (see later)
- No risk of memory leaks
  - see later for double delete's
- Exception safe

Smart pointers

- They behave like pointers thanks to the overloading of `operator*` and `operator->`

```
template<typename Pointee>
class SmartPointer
{
  using Pointer = Pointee*;
  using Reference = Pointee&;
  Pointer p_;
 public:
  // …
  Pointer operator->() { return p_; }
  Reference operator*() { return *p_; }
};

struct X {
  void f();
};

SmartPointer<X> xp(new X);
xp->f();
(*xp).f();
```

Smart pointers and copyability

- What does it mean to copy a (smart) pointer?

```
int main()
{
  SmartPointer<X> p1 {new X};
  SmartPointer<X> p2 {p1};    // copy construction
  SmartPointer<X> p3 {…};
  p3 = p1;                    // copy assignment
}
```

- Do p1 and p2/p3 manage the same pointee?
- After the copy, what about the previous pointee of p2/p3?

Two smart pointers in the standard

- `unique_ptr<T>`, `unique_ptr<T[]>`
  - exclusive ownership
  - movable, non-copyable
  - no space nor runtime overhead

- `shared_ptr<T>` (soon also `shared_ptr<T[]>`)
  - shared ownership
  - movable and copyable
  - some space and runtime overhead (but not for pointer access)

unique_ptr

- unique_ptr<T>, unique_ptr<T[]>
  - exclusive ownership
    - no possibility of double delete
  - movable, non-copyable
  - no space nor runtime overhead

```
struct X {
  int k;
  X(int j): k(j) {}
};

auto use(std::unique_ptr<X> x) -> decltype(x->k) {
  return x->k;
}

int main() {
  auto x = std::make_unique<X>(3);                  // new X{3}
  std::cout << use(x) << '\n';                      // error, not copyable
  std::cout << use(std::make_unique<X>(3)) << '\n'; // ok, movable
}
```

shared_ptr

- shared_ptr<T> (soon also shared_ptr<T[]>)
  - shared ownership (reference counted)
    - the last one to be destroyed will delete → no double delete
  - movable and copyable
  - some space and runtime overhead (only for the management of the reference count)

```
struct X {
  int k;
  X(int j): k(j) {}
};

auto use(std::shared_ptr<X> x) -> decltype(x->k) {
  return x->k;
}

int main() {
  auto x = std::make_shared<X>(3);                    // new X{3}
  std::cout << use(x) << '\n';                        // ok, copyable
  std::cout << use(std::make_shared<X>(3)) << '\n'; // ok, movable
}
```

Access to the raw pointer

- Access to the raw pointer may be needed
  - e.g. to access a legacy API
- `unique/shared_ptr<>::get()`
  - returns a non-owning raw pointer
- unique_ptr<>::release()
  - returns an owning raw pointer
  - must be explicitly managed

Support for a custom deleter

- Smart pointers can be used to manage any resource
  - the resource release is not necessarily done with delete
- Both `unique/shared_ptr` support a custom deleter

```
auto u1 = std::unique_ptr<FILE, decltype(&std::fclose)>{
  std::fopen("/tmp/afile", "r"),
  &std::fclose                                    // int fclose(FILE *fp);
};
auto u2 = std::unique_ptr<FILE, std::function<int(FILE*)>>{
  std::fopen("/tmp/afile", "r"),
  std::fclose
};
auto u3 = std::unique_ptr<FILE, std::function<void(FILE*)>>{
  std::fopen("/tmp/afile", "r"),
  [](FILE* f) { std::fclose(f); }
};
auto u4 = std::shared_ptr<FILE>{
  std::fopen("/tmp/afile", "r"),
  std::fclose
};
```

(cannot use `make_unique` and `make_shared`)

Outline

- An evolving language
  - C++ Standard timeline
  - *C++11 feels like a new language*
- Why pointers?
  - Typical uses
  - Issues with pointers
- Smart pointers
  - How they work
  - How to use them
- Move semantics
- Error management
- Putting it all together

Example: interface to C API

```
DIR *opendir(const char *name);
int closedir(DIR *dirp);

auto dir = std::shared_ptr<DIR>{opendir("/tmp"), closedir};

dirent entry;
for (auto* result = &entry; readdir_r(dir.get(), &entry, &result) == 0 && result; ) {
  std::cout << entry.d_name << '\n';
}
```

- No owning pointers any more
- The only pointer left is `result`, which is non-owning
  - that's fine
- Of course if you need to manipulate directories and files you should use boost.filesystem (soon in std)

Example: pimpl

```
// in state_machine.hh

class StateMachine {
  class Impl;
  using ImplP = std::shared_ptr<Impl>;
  ImplP impl_;
public:
  StateMachine(…);
  void configure();
  void start();
  void stop();
  // …
};
```

```
// in state_machine.cc

class StateMachine::Impl {
  // actual data members
  Impl( … ): … {}
}

StateMachine::StateMachine(…)
  : impl_{make_shared<Impl>( … )}
{ … }

void StateMachine::configure() {
  impl_-> … ;
}
// …
```

- Again, no raw pointers any more
- The default destructor is fine