ESC'15
Bertinoro, 25 – 31 October 2015

Efficient C++ Coding

Francesco Giacomini
INFN

Introduction

- C++ is a complex and large language (and library)
- The following material only scratches the surface
- If you want to know more start from
  - https://isocpp.org/
  - http://en.cppreference.com/
- Continue with the material of the main C++ conferences
  - https://github.com/CppCon
  - https://github.com/boostcon
- And finally you can read the Standard Committee papers
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/
- Some snippets of code are taken from the above sources

Online compilers

- You can edit and try your code online with multiple compilers at
  - http://gcc.godbolt.org/
  - http://coliru.stacked-crooked.com/

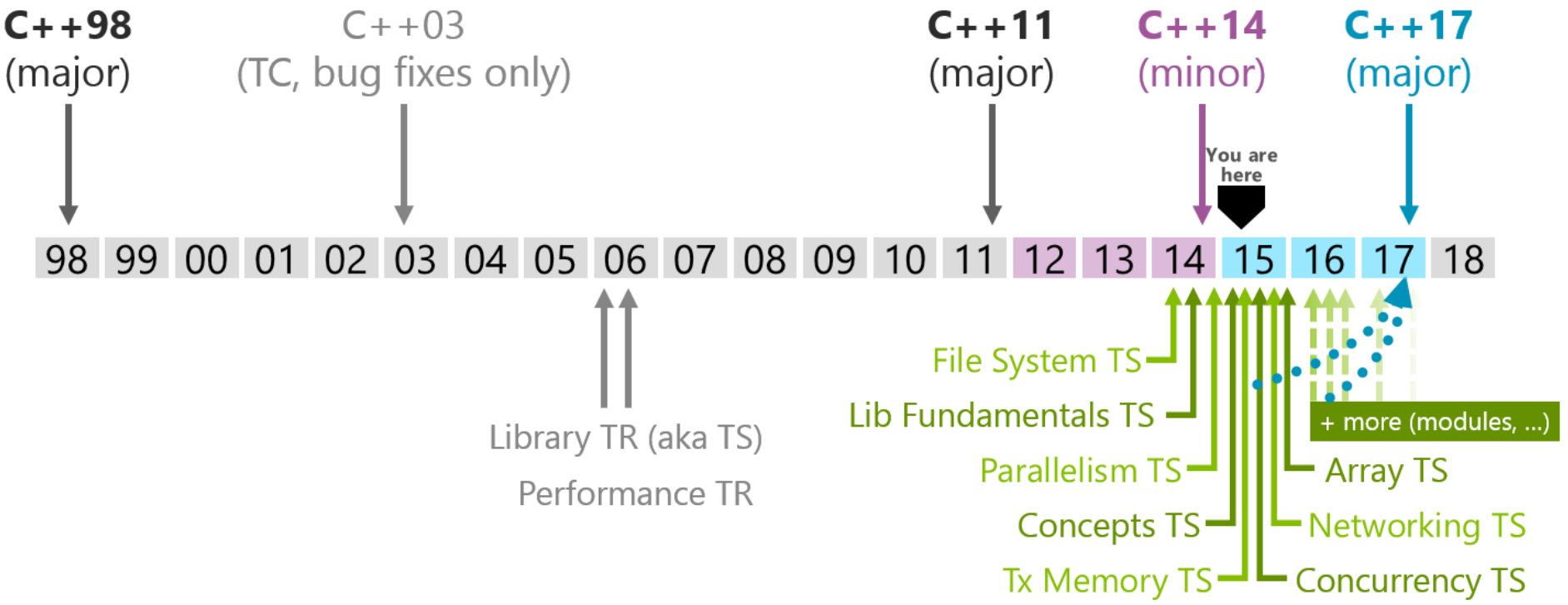
Overview

- An evolving language
- Type deduction
- Function, functor, lambda
- Pointers
- Move semantics
- Contracts

Overview

- An evolving language
- Type deduction
- Function, functor, lambda
- Pointers
- Move semantics
- Contracts

An evolving language



from http://isocpp.org/

C++11 feels like a new language

```
typedef std::map<int, std::string> map_type;

map_type make_map()
{
  map_type result;

  result.insert(std::make_pair(23, "Andrea"));
  result.insert(std::make_pair(49, "Camilla"));
  result.insert(std::make_pair(96, "Ugo"));
  result.insert(std::make_pair(72, "Elsa"));

  return result;
}

int main()
{
  map_type const m = make_map();

  for (map_type::const_iterator it = m.begin();
       it != m.end(); ++it) {
    std::cout << it->first << ' '
              << it->second << '\n';
  }
}
```

C++98

using vs typedef

```
using map_type = std::map<int, std::string>;

auto make_map() -> map_type
{
  map_type const result = {
    {23, "Andrea"},
    {49, "Camilla"},
    {96, "Ugo"},
    {72, "Elsa"}
  };

  return result;
}

int main()
{
  auto const m = make_map();

  for (auto& p: m) {
    std::cout << p.first << ' '
              << p.second << '\n';
  }
}
```

C++11

initializer list

brace-initialization

trailing return type

efficient return-by-value

deduced type

range for

C++11 feels like a new language

```
typedef std::map<int, std::string> map_type;

map_type make_map()
{
  map_type result;

  result.insert(std::make_pair(23, "Andrea"));
  result.insert(std::make_pair(49, "Camilla"));
  result.insert(std::make_pair(96, "Ugo"));
  result.insert(std::make_pair(72, "Elsa"));

  return result;
}

int main()
{
  map_type const m = make_map();

  for (map_type::const_iterator it = m.begin();
       it != m.end(); ++it) {
    std::cout << it->first << ' '
              << it->second << '\n';
  }
}
```

C++98

using vs typedef

```
using map_type = std::map<int, std::string>;

auto make_map()
{
  map_type const result = {
    {23, "Andrea"},
    {49, "Camilla"},
    {96, "Ugo"},
    {72, "Elsa"}
  };

  return result;
}

int main()
{
  auto const m = make_map();

  for (auto& p: m) {
    std::cout << p.first << ' '
              << p.second << '\n';
  }
}
```

C++14

initializer list

brace-initialization

trailing return type

efficient return-by-value

deduced type

range for

Overview

- An evolving language
- Type deduction
- Function, functor, lambda
- Loops
- Pointers
- Move semantics
- Error management

auto

- In

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

why shall we tell the compiler what the type of `iter` is? it must know!

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

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

```
auto v;                     // error, an initializer is required
auto v = 1;                 // v has type int
auto v = 1UL;               // v has type unsigned long
auto v = 1.;                // v has type double
auto v = 1.F;               // v has type float
auto v = 'a';               // v has type char
auto v = "a";               // v has type char const* (C-style string)
auto v = std::string{"a"};  // v has type std::string
```

auto and references

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

```
T v;

auto  v1 = v;  // v1 has type T  - v1 is a copy of v
auto& v2 = v;  // v2 has type T& - v2 is an alias of v
auto  v3 = v2; // v3 has type T  - v3 is a copy of v
```

auto and const

- `auto` makes a mutable copy
- `auto const` makes a non-mutable copy
- `auto&` preserves `const`-ness

```
T v;

auto        v1 = v;  // v1 has type T        - v1 is a mutable copy of v
auto const  v2 = v;  // v2 has type T const  - v2 is a non-mutable copy of v
auto&       v3 = v;  // v3 has type T&       - v3 is a mutable alias of v
auto const& v4 = v;  // v4 has type T const& - v4 is a non-mutable alias of v
```

```
T const v;

auto        v1 = v;  // v1 has type T        - v1 is a mutable copy of v
auto const  v2 = v;  // v2 has type T const  - v2 is a non-mutable copy of v
auto&       v3 = v;  // v3 has type T const& - v3 is a non-mutable alias of v
auto const& v4 = v;  // v4 has type T const& - v4 is a non-mutable alias of v
```

Overview

- An evolving language
- Type deduction
- Function, functor, lambda
- Pointers
- Move semantics
- Contracts

Functions

- C++ entities that associate a sequence of statements (a function body) with a name and a list of zero or more parameters
  - multiple functions can have the same name → overloading
  - the return type is not used for overload resolution

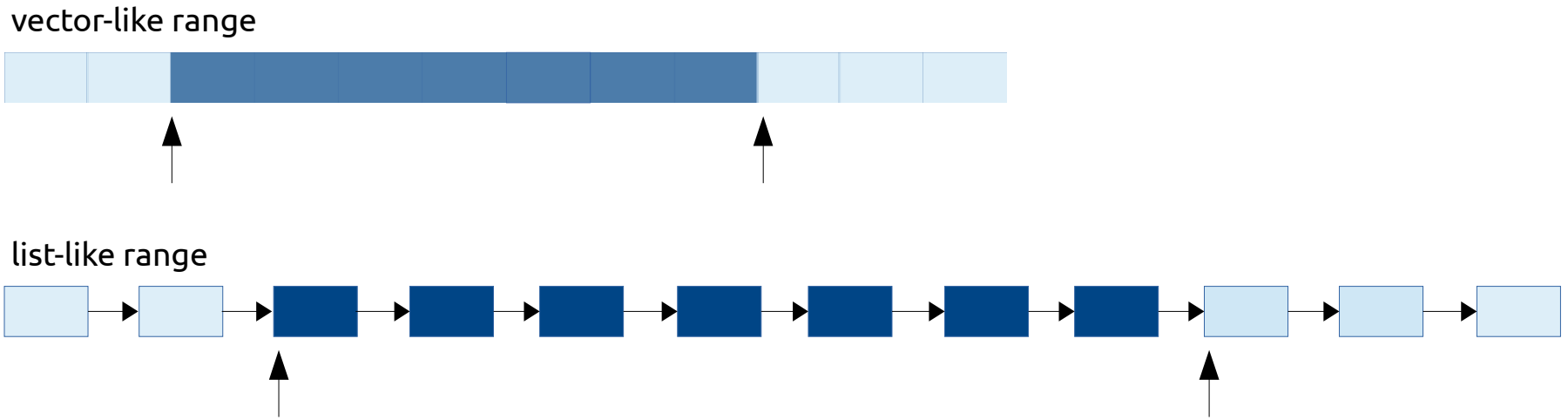
```
std::string cat(std::string const& s, int i); // function declaration

std::string cat(std::string const& s, int i)  // function definition
{
  return s + '-' + std::to_string(i);
}

// cat(std::string{"XYZ"}, 5) returns std::string{"XYZ-5"}
```

- A function returning bool is called a *predicate*

```
bool less(int n, int m)
{
  return n < m;
}
```

Algorithms

- The C++ standard library includes 100+ generic algorithms, in the form of function templates
- Defined in `<algorithm>` or `<numeric>`
- Algorithms operate on ranges, represented as a pair of iterators pointing to the beginning and one past the end of the range



Algorithms

- Examples

```
std::vector<int> v = { 2, 5, 4, 0, 1 };

// sort the first half of the vector in ascending order
std::sort(std::begin(v), std::begin(v) + v.size() / 2);

// sum up the vector elements, initializing the sum to 0
auto s = std::accumulate(std::begin(v), std::end(v), 0);

// append the partial sums of the vector elements into a list
std::list<int> l;
std::partial_sum(std::begin(v), std::end(v), std::back_inserter(l));

// find the first element with value 4
auto it = std::find(std::begin(v), std::end(v), 4);
```

- Some algorithms are customizable passing a function

```
// sum up the vector elements using the cat function, starting from an empty string
auto s = std::accumulate(std::begin(v), std::end(v), std::string{}, cat);
```

Functors

- Another mechanism to define *something-callable-like-a-function*
  - a class with an `operator()`

```
auto cat(std::string const& s, int i)
{
  return s + '-' + std::to_string(i);
}

auto s = cat(std::string{"XYZ"}, 5); // XYZ-5
```

```
struct Cat
{
  auto operator()(std::string const& s, int i) const
  {
    return s + '-' + std::to_string(i);
  }
};

Cat cat;
auto v = cat(std::string{"XYZ"}, 5);    // XYZ-5

// a Cat built on-the-fly
auto v = Cat{}(std::string{"XYZ"}, 5);  // XYZ-5

auto v = std::accumulate(std::begin(v), std::end(v),
                         std::string{}, cat);
auto v = std::accumulate(std::begin(v), std::end(v),
                         std::string{}, Cat{});
```

Functors

- A functor, being a class, can have state

```
class CatWithState
{
 public:
  explicit Cat(char c): c_(c) {}
  auto operator()(std::string const& s, int i) const
  {
    return s + c_ + std::to_string(i);
  }
 private:
  char c_;
};

CatWithState cat1{'-'};
auto v = cat1(std::string{"XYZ"}, 5);              // XYZ-5

CatWithState cat2{'%'};
auto v = cat2(std::string{"XYZ"}, 5);              // XYZ%5

// a CatWithState built on-the-fly
auto v = CatWithState{'^'}(std::string{"XYZ"}, 5); // XYZ^5

CatWithState cat3{'_'};
auto v = std::accumulate(std::begin(v), std::end(v), std::string{}, cat3);

// pass a CatWithState built on-the-fly
auto v = std::accumulate(std::begin(v), std::end(v), std::string{}, CatWithState{'_'});
```

Functors

- An example from the standard library

```
#include <random>

// random number generator (mersenne twister)
std::mt19337 gen;

// generate N 32-bit unsigned integer numbers

for (int n = 0; n != N; ++n) {
  std::cout << gen() << '\n';
}

// generate N doubles distributed normally (mean: 0., stdev: 1.)

std::normal_distribution<> dist;

for (int n = 0; n != N; ++n) {
  std::cout << dist(gen) << '\n';
}

// generate N ints distributed uniformly

std::uniform_int_distribution<> roll_dice(1, 6);

for (int n = 0; n != N; ++n) {
  std::cout << role_dice(gen) << '\n';
}
```

Lambda

- A concise way to create anonymous functors
- Useful to pass actions/callbacks to algorithms, threads, frameworks, …

```
struct Cat {
  auto operator()(std::string const& s, int i) const
  {
    return s + '-' + std::to_string(i);
  }
};


class CatWithState
{
 public:
  explicit CatWithState(char c): c_(c) {}
  auto operator()(std::string const& s, int i) const
  {
    return s + c_ + std::to_string(i);
  }
 private:
  char c_;
};
```

```
auto s = std::accumulate(..., Cat{});

auto s = std::accumulate(...,
    [](std::string const& s, int I) {
      return s + '-' + std::to_string(i);
    }
);

auto s = std::accumulate(..., CatWithState{'-'});

char c{'-'};
auto s = std::accumulate(...,
    [=](std::string const& s, int I) {
      return s + c + std::to_string(i);
    }
);

auto s = std::accumulate(...,
    [=](auto const& s, auto I) {
      return s + c + std::to_string(i);
    }
);
```

Lambda

- The evaluation of a lambda expression produces a closure, containing
  - the code of the body of the lambda
  - the *environment* in which the lambda is defined
    - the variables that are referenced in the body are stored in the generated functor

C++11

```
int v = 3;

auto l = [=](int i) {
    return i + v;
};

int r = l(5); // r == 8
```

C++14

```
auto l = [v = 3](int i) {
    return i + v;
};

int r = l(5); // r == 8
```

is transformed to

```
int v = 3;

class SomeUniqueName {
 public:
  explicit SomeUniqueName(int v): v_(v) {}
  auto operator()(int i) const {
    return i + v_;
  }
 private:
  int v_;
};

SomeUniqueName l(v);
int r = l(5);        // r == 8
```

Lambda

- Automatic variables used in the lambda body need to be captured
  - `[]` capture nothing
  - `[=]` capture all by value
  - `[&]` capture all by reference
  - `[=, &k]` capture all by value but `k` by reference
  - `[&, k]` capture all by reference but `k` by value
- Variables captured by reference can be modified
  - there is no way to capture by const reference

```
int v = 3;
[&]{ ++v; }();
assert(v == 4);
```

Lambda

- By default the call to a lambda is `const`
  - variables captured by value cannot be modified
    - because local copies would be modified
  - a lambda can be set to be `mutable`

```
int v = 3;
[=]() mutable { ++v; }();
assert(v == 3);
```

- Global/static variables are available without being captured

Lambda

Be careful not to have dangling references

Compare this...

```
int GOOD_make_int()
{
  int v = 42;
  return v;
}

int j = GOOD_make_int(); // Ok

int& BAD_make_int()
{
  int v = 42;
  return v;
}

int j = BAD_make_int(); // BUM! (maybe)
```

… to this

```
auto GOOD_make_lambda()
{
  int v = 42;
  return [=] { return v; };
}

auto j = GOOD_make_lambda()(); // Ok

auto BAD_make_lambda()
{
  int v = 42;
  return [&] { return v; };
}

auto j = BAD_make_lambda()(); // BUM! (maybe)
```

The compiler is not always helpful

Lambda

- How big is a lambda?

```
std::array<int, 10> y;
auto l = [=]() { return y.size(); };
static_assert(sizeof(l) == sizeof(int) * y.size());      // 40 (4 * 10)

auto l = [&]() { return y.size(); };
static_assert(sizeof(l) == sizeof(std::array<int, 10>*); // 8
```

std::function

- *Type-erased* wrapper that can store and invoke any callable entity with a certain signature
  - function, functor, lambda, member function
- Some space and time overhead, so use only if a template parameter is not satisfactory

```
#include <functional>
#include <vector>
#include <iostream>

int sum_squares(int x, int y) { return x * x + y * y; }

int main()
{
  std::vector<std::function<int (int, int)>> v;
  v.emplace_back(std::plus<>());       // plus has a compatible operator()
  v.emplace_back(std::multiplies<>()); // idem
  v.emplace_back(&sum_squares);
  for (int k = 10; k <= 1000; k *= 10) {
    v.emplace_back([k](int x, int y) {return i * x + y; }); // implicit `-> int`
  }
  for (const auto& f : v) {
    std::cout << f(4, 5) << '\n';
  }
}
```

Hands-on

- C++ → Algorithms
- Starting from `algo.cpp`, write code to
  - sum all the elements of the vector
  - multiply all the elements of the vector
  - sort the vector in ascending and descending order
  - move the even numbers at the beginning
  - move the three central numbers at the beginning
  - erase from the vector the elements that satisfy a predicate
  - …

Overview

- An evolving language
- Type deduction
- Function, functor, lambda
- Pointers
- Move semantics
- Contracts

What's wrong with T*

- 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 T*

- Am I the owner of the pointee (the object pointed to)? who is responsible for the delete?
- Explicit allocation and deallocation increases the risk of memory leaks and double deletes

```
{
  T* t = new T(...);
  …
  // ops, forgot to call delete t
}
{
  T* t = new T(...);
  …
  delete t;
  …
  delete t; // ops, delete again
}
```

What's wrong with T*

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

```
{
  T* t = new T(...);
  // potentially throwing code
  delete t; // not executed in case an exception is thrown
}
```

What's wrong with T*

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

```
void f()
{
  int m{123};
}

void g()
{
  int* m = new int{123};
  delete m;
}
```

compiled into*

*slightly edited

```
f():
     subq %4, %rsp
     movl $123, (%rsp)
     addq $4, %rsp
     ret
g():
     subq $8, %rsp
     movl $4, %edi
     call operator new(unsigned long)
     movl $123, (%rax)
     movl $4, %esi
     movq %rax, %rdi
     call operator delete(void*, unsigned long)
     addq $8, %rsp
     ret
```

What's wrong with T*

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

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

When to use a T*

- To represent a *link* to an object when
  - the object is not owned, and
    - the link may be null, or
    - the link can be re-bound
- Mutable and immutable scenarios

```
T* tp = nullptr;
```

```
T* tp = &t1;
tp = &t2;
```

```
void f(T* tp)
{
  if (!tp) {
    // check if tp is null
  }

  // can read and modify *tp

  // do not call 'delete tp'
}
```

```
void f(T const* tp)
{
  if (!tp) {
    // check if tp is null
  }

  // can only read *tp

  // do not call 'delete tp'
}
```

When **not** to use a T*

- To represent a link to an object when
  - the object is owned, or
    - the link can never be null, and
    - the link cannot be re-bound
- Alternatives
  - use a (const) reference

```
T& tr = t;
tr = t2;          // error: cannot re-bind
T const& tcr = t;
```

compare with

```
T* tp = nullptr;
tp = &t;
```

  - use a copy

```
T tr = t;
tr = t2;          // ok
T const tcr = t;
```

  - use a smart pointer

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
- Small (if any) space and time overhead w.r.t. a raw pointer
- No risk of memory leaks or double delete's
- Exception safe

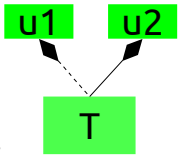
Smart pointers

```
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();
```

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

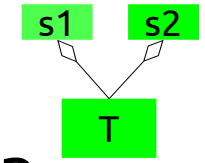
Two smart pointers in the standard

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




```
struct X {
  X(T1 t1, T2 t2);
  ...
};

void use(std::unique_ptr<X> x);

{
  // std::unique_ptr<X> x(new X(t1, t2));
  auto x = std::make_unique<X>(t1, t2);
  use(x);             // error, not copyable
  use(std::move(x)); // ok, movable
}
```

- `shared_ptr<T>`
  - shared ownership (reference counted)
  - some space and time overhead
    - for the management, but **not for access**
  - copyable and movable




```
void use(std::shared_ptr<X> x);

{
  // std::shared_ptr<X> x(new X(t1, t2));
  auto x = std::make_shared<X>(t1, t2);
  use(x);             // ok, copyable
  use(std::move(x)); // ok, movable
}
```

Prefer `unique_ptr` unless you need `shared_ptr`

Access to the raw T*

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

Custom deleter

- Consider a smart pointer as a general-purpose resource handler
  - the resource release is not necessarily done with `delete`
- Both `unique/shared_ptr<>` support a custom deleter

```
FILE* f = fopen("/tmp/afile", "r");
…
int e = fclose(f);
```

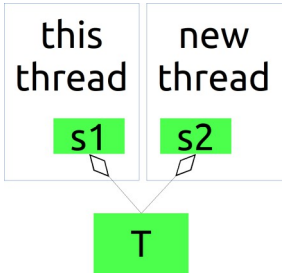
Usual problems

- who owns the resource
- forgetting release
- early return/throw
- double release

```
auto u = std::shared_ptr<FILE>{
  std::fopen("/tmp/afile", "r"),
  std::fclose
};

auto u = std::unique_ptr<FILE, decltype(&std::fclose)>{
  std::fopen("/tmp/afile", "r"),
  &std::fclose
};

auto u = std::unique_ptr<FILE, std::function<int(FILE*)>>{
  std::fopen("/tmp/afile", "r"),
  std::fclose
};

auto u = std::unique_ptr<FILE, std::function<void(FILE*)>>{
  std::fopen("/tmp/afile", "r"),
  [](FILE* f) { std::fclose(f); }
};
```

Smart pointers and functions

- Pass a smart pointer to a function only if the function needs to rely on the smart pointer itself
  - by value to keep the resource alive




```
auto s1 = make_shared<T>();
std::thread t([s2 = s1]{ do_something(*s2); })
```

  - by ref to interact with the smart pointer itself

```
void print_count(shared_ptr<T> const& s) { cout << s.use_count() << '\n'; }
auto s = make_shared<T>();
print_count(s);
```

  - to transfer ownership (for `unique_ptr`)

```
void take(std::unique_ptr<S> u) { … }
auto u = std::make_unique<S>();
take(u);              // fail to compile
take(std::move(u)); // ok
```

- Otherwise pass the pointed-to object by (const) ref/ptr

```
void f(shared_ptr<T> s) { s->g(); }
void f(T* t)            { if (t) t->g(); } // better
void f(T& t)            { t.g(); }         // better
```

```
auto s = make_shared<S>();
f(s);
f(s->get());
f(*s);
```

Smart pointers and functions

- Return a smart pointer from a function if the function has dynamically allocated a resource that is passed to the caller
  - prefer to return a `unique_ptr`
  - if needed the `unique_ptr` can transfer ownership to a `shared_ptr`
    - but not viceversa

```
std::unique_ptr<T> factory()
{
  return std::make_unique<T();
}

auto u = factory();               // u has type unique_ptr<T>
std::shared_ptr<T> s = factory(); // a unique_ptr is convertible to a shared_ptr
```

Hands-on

- C++ → Pointers
- Starting from `dir.cpp`, write code to:
  - create a smart pointer managing a DIR resource obtained with the opendir() function call
  - associate a deleter to that smart pointer
  - implement a function to read the names of the files in that directory
  - check if the deleter is called at the right moment
  - hide the creation of the smart pointer behind a factory function
  - …

Overview

- An evolving language
- Type deduction
- Function, functor, lambda
- Pointers
- Move semantics
- Contracts

Return a value from a function

- Returning a large value from a function is often perceived as “slow”
  - return “by pointer”

```
std::unique_ptr<LargeObject> make_large_object() {
  return std::make_unique<LargeObject>{};
}

auto lo = make_large_object();
lo-> … ; // use the object, via a pointer
```

  - use “out” function arguments

```
void make_large_object(LargeObject& o) {
  o = LargeObject{};   // requires copy assignment
}

LargeObject lo;        // requires default constructor
make_large_object(lo);
lo. … // use the object
```

Return a value from a function

- Wouldn't it be better to write

```
LargeObject make_large_object() {
  return LargeObject{};           // requires copy assignment
}

auto lo = make_large_object();    // or auto const lo = …
lo. …                             // use the object
```

  ?

- In fact the compiler is allowed to elide the copy of the returned value into the final destination
  - (N)RVO – (Named) Return Value Optimization
- If (N)RVO is not applied, C++11 mandates a move, if available
- If the move is not available, copy; this may be really expensive

Return Value Optimization

- Unnamed

```
Type make_urvo(…)
{
  if (…) {
    return Type(…);
  }
  return Type(…);
}

auto t = make_urvo(…);
```

- Named

```
Type make_nrvo(…)
{
  Type result;
  if (…) {
    result = Type(…);
    return result;
  }
  return result;
}

auto t = make_nrvo(…);
```

- Try not to mix URVO e NRVO
  - but it may still be ok if Type is cheaply movable
- Don't

```
return std::move(result);
```

```
Type make_no_rvo(…)
{
  Type result;
  if (…) {
    return Type(…);
  }
  return result;
}

auto t = make_no_rvo(…);
```

Copy vs move

```
String s1("...");
String s2(s1);           // (1)

String get_string();
String s3(get_string()); // (2)
```

- In (1) the deep copy is necessary, because `s1` still exists after the copy
- In (2) the deep copy is a waste, because the temporary created by `get_string()` is immediately destroyed after the construction of `s3`
- Named objects are called lvalues
  - and you can take their address
- Temporary (unnamed) objects are called rvalues
  - and you can't take their address

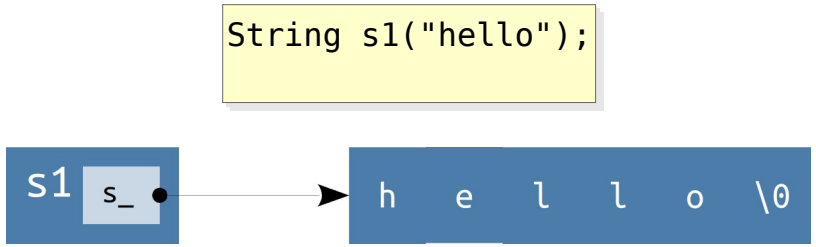
```
// (2) is roughly equivalent to

String tmp = get_string();
String s3 = tmp;
tmp.~String();
```

Copy vs move

```
String s1("hello");
```

Copy vs move

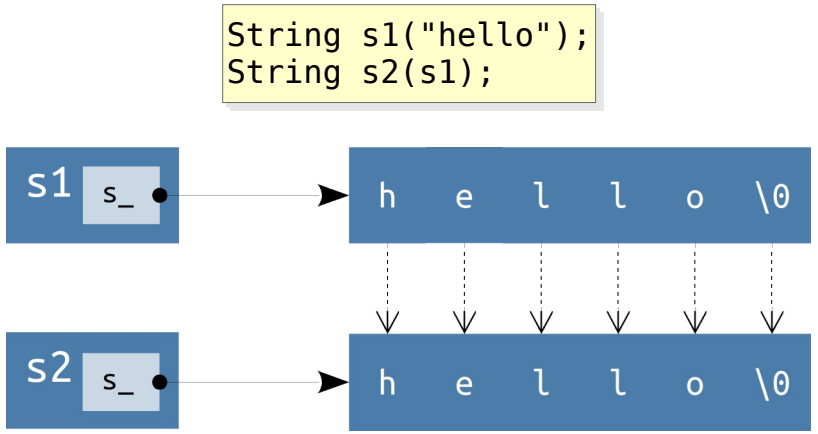
```
class String {
  char* s_; // nullptr or null-terminated*
 public:
  String(char const* s) {
    size_t size = strlen(s) + 1;
    s_ = new char[size];
    memcpy(s_, s, size);
  }
  ~String() { delete [] s_; }









  size_t size() const { return strlen(s_); }
};
```



* not using a smart pointer is for illustrative purposes only

Copy vs move

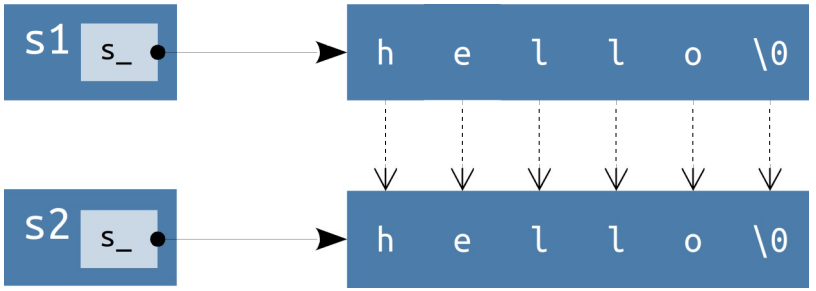
```
class String {
  char* s_; // nullptr or null-terminated
 public:
  String(char const* s) {
    size_t size = strlen(s) + 1;
    s_ = new char[size];
    memcpy(s_, s, size);
  }
  ~String() { delete [] s_; }
  // copy
  String(String const& other) {
    size_t size = strlen(other.s_) + 1;
    s_ = new char[size];
    memcpy(s_, other.s_, size);
  }



  size_t size() const { return strlen(s_); }
};
```



Copy vs move

```
class String {
  char* s_; // nullptr or null-terminated
 public:
  String(char const* s) {
    size_t size = strlen(s) + 1;
    s_ = new char[size];
    memcpy(s_, s, size);
  }
  ~String() { delete [] s_; }
  // copy
  String(String const& other) {
    size_t size = strlen(other.s_) + 1;
    s_ = new char[size];
    memcpy(s_, other.s_, size);
  }




  size_t size() const { return strlen(s_); }
};

String get_string();
```

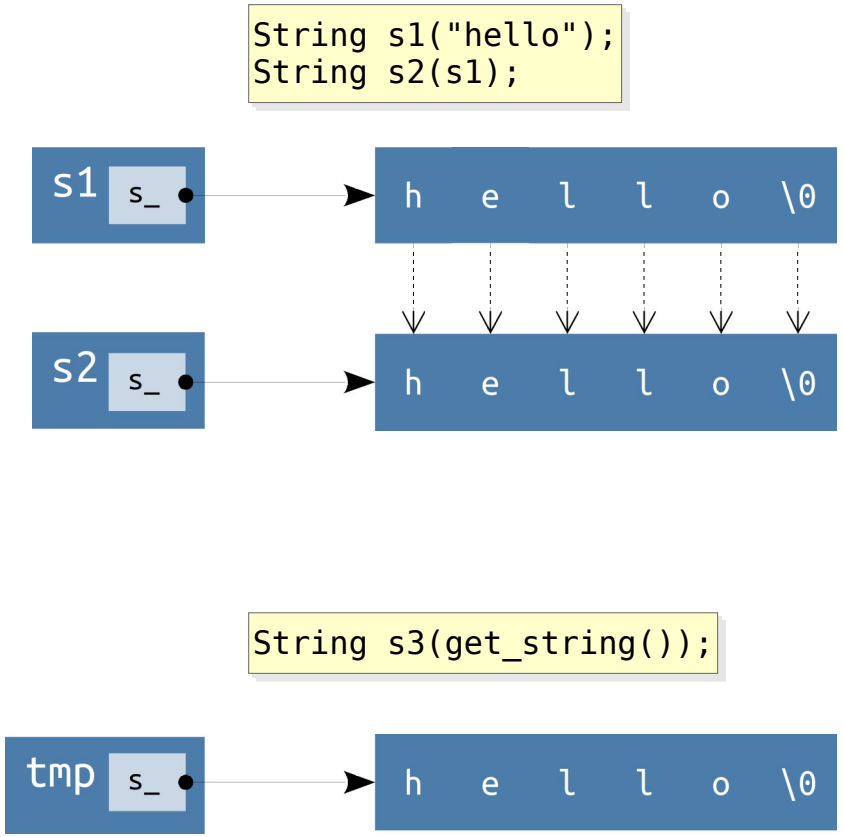
```
String s1("hello");
String s2(s1);
```




```
String s3(get_string());
```

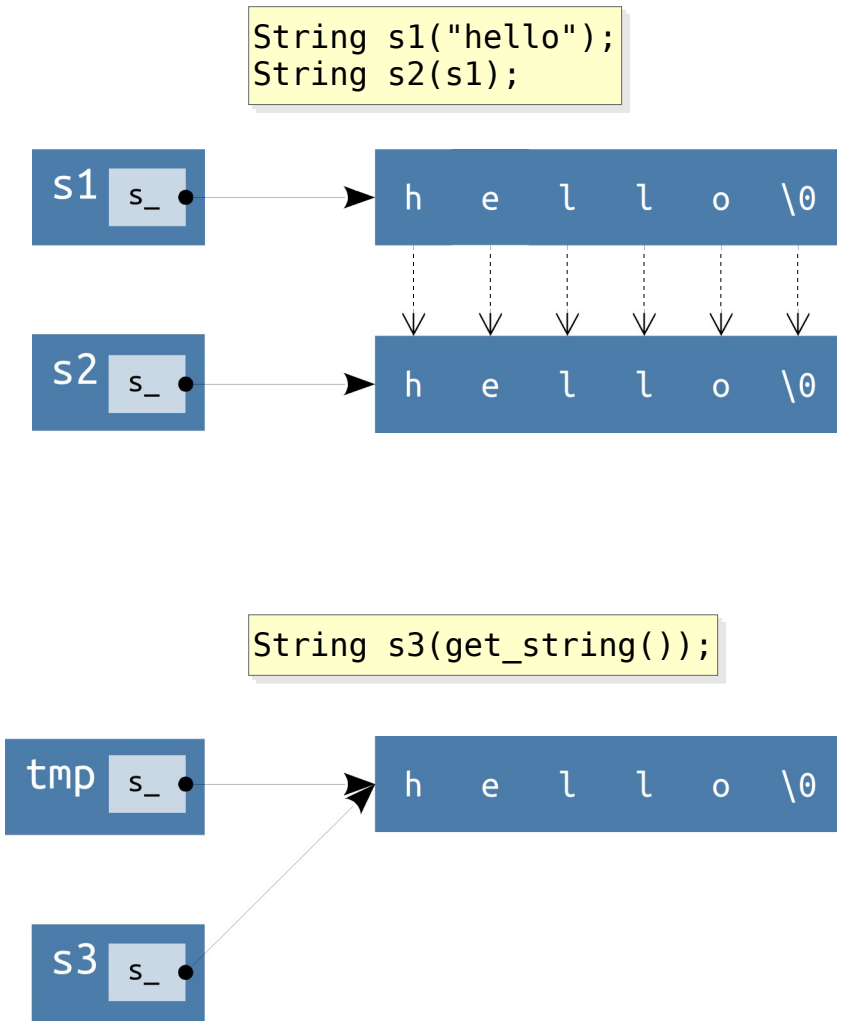
Copy vs move

```
class String {
  char* s_; // nullptr or null-terminated
 public:
  String(char const* s) {
    size_t size = strlen(s) + 1;
    s_ = new char[size];
    memcpy(s_, s, size);
  }
  ~String() { delete [] s_; }
  // copy
  String(String const& other) {
    size_t size = strlen(other.s_) + 1;
    s_ = new char[size];
    memcpy(s_, other.s_, size);
  }




  size_t size() const { return strlen(s_); }
};

String get_string();
```



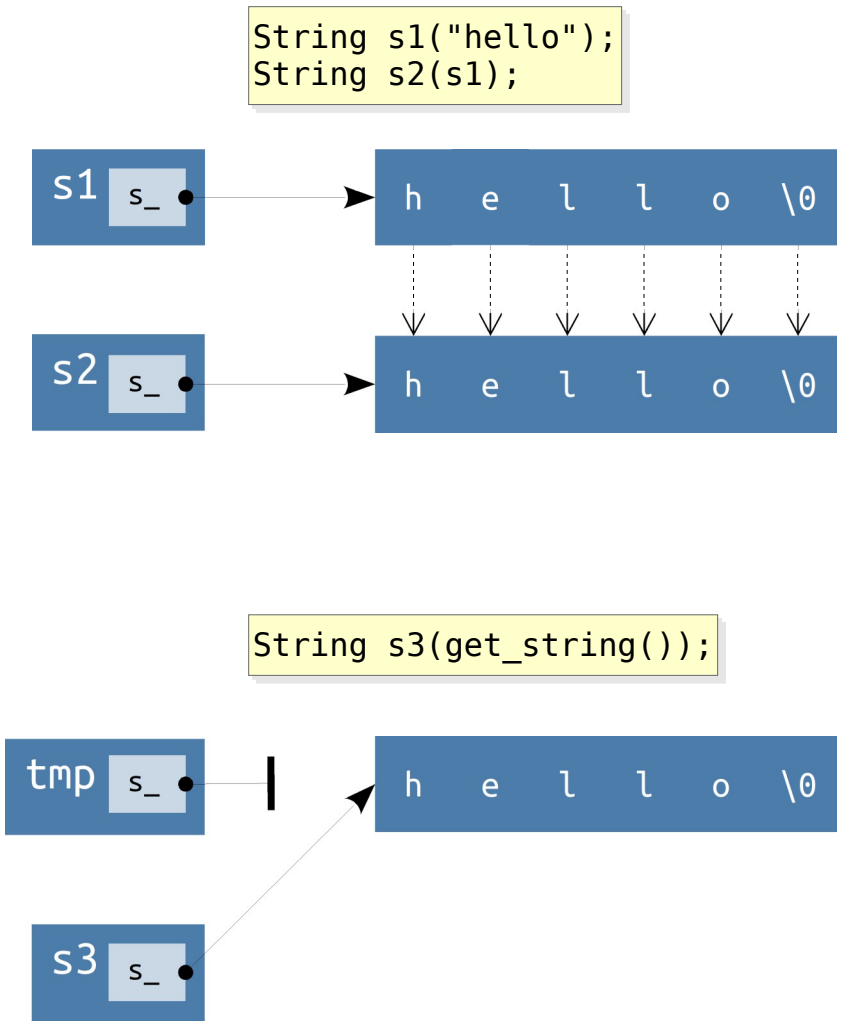
Copy vs move

```
class String {
  char* s_; // nullptr or null-terminated
 public:
  String(char const* s) {
    size_t size = strlen(s) + 1;
    s_ = new char[size];
    memcpy(s_, s, size);
  }
  ~String() { delete [] s_; }
  // copy
  String(String const& other) {
    size_t size = strlen(other.s_) + 1;
    s_ = new char[size];
    memcpy(s_, other.s_, size);
  }
  // move
  String(??? tmp): s_(tmp.s_) {

  }
  size_t size() const { return strlen(s_); }
};

String get_string();
```



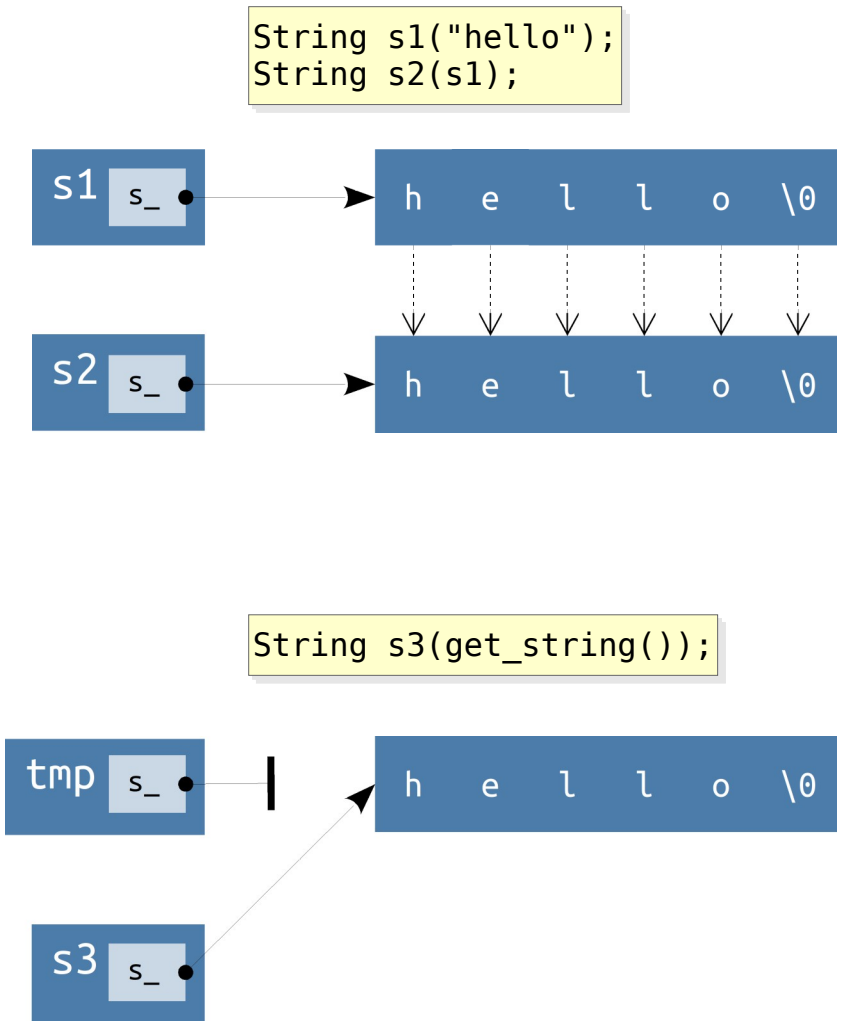
Copy vs move

```
class String {
  char* s_; // nullptr or null-terminated
 public:
  String(char const* s) {
    size_t size = strlen(s) + 1;
    s_ = new char[size];
    memcpy(s_, s, size);
  }
  ~String() { delete [] s_; }
  // copy
  String(String const& other) {
    size_t size = strlen(other.s_) + 1;
    s_ = new char[size];
    memcpy(s_, other.s_, size);
  }
  // move
  String(??? tmp): s_(tmp.s_) {
    tmp.s_ = nullptr;
  }
  size_t size() const { return strlen(s_); }
};

String get_string();
```



Copy vs move

```
class String {
  char* s_; // nullptr or null-terminated
 public:
  String(char const* s) {
    size_t size = strlen(s) + 1;
    s_ = new char[size];
    memcpy(s_, s, size);
  }
  ~String() { delete [] s_; }
  // copy
  String(String const& other) {
    size_t size = strlen(other.s_) + 1;
    s_ = new char[size];
    memcpy(s_, other.s_, size);
  }
  // move
  String(??? tmp): s_(tmp.s_) {
    tmp.s_ = nullptr;
  }
  size_t size() const { return strlen(s_); }
};

String get_string();
```



rvalue reference

- In C++98 there is no way to overload functions specifically for rvalue (i.e. temporary) arguments
- C++11 introduced *rvalue references* (T&&)

```
class String
{
  // move constructor
  String(String&& tmp): s_(tmp.s_) { tmp.s_ = nullptr; }
};

String s1;
String s2(s1);           // call String::String(String const&)
String s3(get_string()); // call String::String(String&&)
```

- Any function can accept rvalue references
- lvalues can be explicitly transformed into rvalues

```
void foo(String&&);

foo(get_string());    // ok
foo(String{"hello"}); // ok
```

```
String s;
foo(s);              // error
foo(std::move(s));   // ok - I don't care about s anymore
s.size();            // undefined
```

Overloading on &&

- A function can be overloaded for temporaries
  - useful if there are significant opportunities of optimization

```
void foo(Type const&) { … }
void foo(Type&&) { … }

Type t{ … };
foo(t);          // calls foo(Type const&)
foo(Type{ … }); // calls foo(Type&&)
```

- For more than one parameter it becomes less desirable
  - consider pass by value, if move is cheap
  - especially useful for sinks, e.g. constructors

```
struct {
  T1 t1_; T2 t2_;
  S(T1 t1, T2 t2) : t1_(std::move(t1)), t2_(std::move(t2)) { … }
};

T1 t1; T2 t2;
S s(t1, make_t2());
S s(make_t1(), t2);
```

Special member functions

- Since C++11 a class has 5 special member functions
  - plus the default constructor

```
class C
{
  T(T const&);             // copy constructor
  T& operator=(T const&); // copy assignment
  T(T&&);                  // move constructor, new in C++11
  T& operator=(T&&);       // move assignment, new in C++11
  ~T();                    // destructor
};
```

- The compiler can generate them automatically under certain constraints
- Rule of thumb: if you need to declare any one of these functions, declare them all
  - consider `= delete`, `= default`

```
class C // movable but non-copyable
{
  T(T const&) = delete;
  T& operator=(T const&) = delete;
  T(T&&) = default;
  T& operator=(T&&) = default;
  ~T() = default;
};
```

= delete

- Prevent the compiler from implicitly generating functions not explicitly declared

C++98

```
class NC
{
private:
  NC(NC const&);
  NC& operator=(NC const&);
};

NC c;
NC c1(c);       // error (at compile or link time)
NC c2; c2 = c;  // error (at compile or link time)
```

C++11

```
class NC
{
private: // or even public
  NC(NC const&) = delete;
  NC& operator=(NC const&) = delete;
};

NC c;
NC c1(c);       // error (at compile time)
NC c2; c2 = c;  // error (at compile time)
```

- The mechanism is applicable to any function

```
void f(double);

f(1.); // ok
f(1);  // ok, calls f passing double(1)

void f(int) = delete;

f(1);  // error

template<class T> void empty(T t) { }
template<> void empty<float>(float t) = delete;

empty(1.f) // error
```

= default

- Explicitly tell the compiler that the default implementation of a special member function is ok
  - To force the generation of the function
  - As a documentation feature

```
// tell with comments
struct C
{
// the implicitly-generated default constructor,
// copy constructor and copy assignment operator
// are ok
};

C c;
C c1(c);
C c2; c2 = c;
```

```
// tell with code
struct C
{
  C() = default;
  C(C const&) = default;
  C& operator=(C const&) = default;
};

C c;
C c1(c);
C c2; c2 = c;
```

Hands-on

- C++ → Move
- Starting from the existing `string.cpp`, write code to
  - Complete the set of the special member functions so that `String` is copyable and movable
  - Add `operator[]` (const and non-const) to access a character at a given position
  - Add a `c_str()` member function to access the underlying C-style string
  - Use a smart pointer instead of a raw pointer
    - note that `std::unique/shared_ptr` and corresponding `make_unique/shared` support arrays
  - …

Overview

- An evolving language
- Type deduction
- Function, functor, lambda
- Pointers
- Move semantics
- Contracts

Mechanisms for error management

The sooner errors are identified, the better

- `static_assert`
  - logical assertion that must be valid at compile time
- `assert`
  - logical assertion that by design must be valid at run time
- Exceptions
  - to express an error condition happening at run time, typically related to lack of a resource
- C-style return codes
  - can be ignored (but they should not!)

static_assert

- Check that a certain constant boolean expression is satisfied during compilation
  - if not, fail compilation with the specified message

```
#include <type_traits>

class NC
{
  NC(NC const&) = delete;
  NC& operator=(NC const&) = delete;
};

static_assert(!std::is_default_constructible<NC>::value, "not default constructible");
static_assert(!std::is_copy_constructible<NC>::value, "not copy constructible");
static_assert(!std::is_copy_assignable<NC>::value, "not copy assignable");
```

- A static assertion declaration can appear practically anywhere in the code
- There is no effect, hence no overhead, at run time

assert

- Check that a certain boolean expression is satisfied during run time
  - if not satisfied, it means that the state of the program is corrupted → better to `terminate` as soon as possible

```
template<class T, … > class vector
{
  T* p;
  …
  T& operator[](size_type n) {
    assert(n < size());
    return p[n];
  }
};
```

- Useful during testing/debugging
  - can be disabled for performance reasons (`-DNDEBUG`)
  - avoid side effects in `assert`'s

Exceptions

- Mechanism to report errors out of a function, stopping its execution
- Cannot be ignored
- Help separate application logic from error management

```
class E {…};
class S {…};

S make_s() {
  auto r = acquire_resources_to_build_s();
  if (!success(r)) {
    E e{…};
    throw e; // or directly throw E{…};
  }
  S result{r};   // not executed in case an
  return result; // exception is thrown
}
```

```
void make_and_use_s() {
  try {
    // …
    auto s = make_s();
    use(s);
    // …
  } catch (E& e) {
    // manage e
  }
}
```

Exceptions

- An exception is propagated up the stack of function calls until a suitable `catch` clause is found
- If no suitable `catch` clause is found the program is `terminated`
- During stack unwinding all object destructors are called
  - remember smart pointers

```
void low() {
  // this part is executed
  throw E{};
  // this part is not executed
}

void mid() {
  T t; // this part is executed
  low();
  // this part is not executed,
  // but ~T() is called
}

void high() {
  try {
    // this part is executed
    mid();
    // this part is not executed
  } catch (E& e) { // by reference
    // use e
  }
}
```

Exception safety

- Different levels of safety guarantees (for member functions):
  - **Basic** - if an exception is thrown, the object is left in a valid, but unspecified, state and no resource is leaked
    - the object should be at least safely assignable and destroyable
    - every class should provide at least the basic guarantee
  - **Strong** – transaction semantics: if an exception is thrown, the object's state is as it was before the function was called
  - **No-throw** – the operation is always successful and no exception leaves the function

noexcept

- A function can be declared `noexcept`, telling the compiler that the function
  - doesn't throw, or
  - is not able to manage possible exceptions → better `terminate`
- What `String` functions can be `noexcept`?

```
class String {
  char* s_; // nullptr or null-terminated
 public:
  String()
    : s_(nullptr)
  {}
  String(char const* s) {
    size_t size = strlen(s) + 1;
    s_ = new char[size];
    memcpy(s_, s, size);
  }
  ~String() {
    delete [] s_;
  }
  String(String const& other) {
    size_t size = strlen(other.s_) + 1;
    s_ = new char[size];
    memcpy(s_, other.s_, size);
  }
  String(String&& tmp)
    : s_(tmp.s_) {
    tmp.s_ = nullptr;
  }
  size_t size() const {
    return s_ ? strlen(s_) : 0;
  }
  …
};
```

noexcept

- A function can be declared noexcept, telling the compiler that the function
  - doesn't throw, or
  - is not able to manage possible exceptions → better `terminate`
- What `String` functions can be noexcept?

```
class String {
  char* s_; // nullptr or null-terminated
 public:
  String() noexcept
    : s_(nullptr)
  {}
  String(char const* s) {
    size_t size = strlen(s) + 1;
    s_ = new char[size];
    memcpy(s_, s, size);
  }
  ~String() noexcept {
    delete [] s_;
  }
  String(String const& other) {
    size_t size = strlen(other.s_) + 1;
    s_ = new char[size];
    memcpy(s_, other.s_, size);
  }
  String(String&& tmp) noexcept
    : s_(tmp.s_) {
    tmp.s_ = nullptr;
  }
  size_t size() const noexcept {
    return s_ ? strlen(s_) : 0;
  }
  …
};
```

noexcept

- A function can be declared noexcept, telling the compiler that the function
  - doesn't throw, or
  - is not able to manage possible exceptions → better `terminate`
- What `String` functions can be noexcept?
- NB: C++11 deprecated dynamic exception specifications

```
class String {
  char* s_; // nullptr or null-terminated
 public:
  String() noexcept
    : s_(nullptr)
  {}
  String(char const* s) {
    size_t size = strlen(s) + 1;
    s_ = new char[size];
    memcpy(s_, s, size);
  }
  ~String() noexcept {
    delete [] s_;
  }
  String(String const& other) {
    size_t size = strlen(other.s_) + 1;
    s_ = new char[size];
    memcpy(s_, other.s_, size);
  }
  String(String&& tmp) noexcept
    : s_(tmp.s_) {
    tmp.s_ = nullptr;
  }
  size_t size() const noexcept {
    return s_ ? strlen(s_) : 0;
  }
  …
};
```

The importance of being noexcept

- Consider

```
class String
{
  // …
  String(String&& tmp) // noexcept
  { … }
};

String s { … };
std::vector<String> v(10000000, s);
v.push_back(s);        // causes re-allocation of the whole vector
```

- How much does noexcept affect the performance?
  - without noexcept: 1077 ms
  - with noexcept: 52 ms
- If the move can throw, `vector` will copy, not move, data
  - push_back provides the strong guarantee

Move and noexcept

- If move operations, especially the constructor, are noexcept the compiler can apply significant optimizations
- `T& T::operator=(T&& tmp)` is typically easy to make `noexcept`
  - rely on `noexcept`-ness of move-assigning all data members
- `T::T(T&& tmp)` may be more difficult
  - start with one object (`tmp`), end up with two (`*this` and `tmp`)
  - can rely on `T::T()` being `noexcept` as well
    - which is not obvious if a resource has to be acquired
    - that's why we don't allocate in `String::String()`

Destructors and noexcept

- Since C++11 the destructor is by default noexcept
  - i.e. releasing a resource should not fail
- Do not do anything overly complicated in the destructor or swallow all exceptions locally

```
class X
{
  ~X() {
    try {
      …
    } catch (...) { // catch all exceptions
      // e.g. log something, provided logging doesn't throw
    }
  }
};
```

- Technically an incompatibility wrt to C++98
- It's always possible to declare a destructor, like any other function, `noexcept(false)`

Error management in practice

- `static_assert`
- Require at compile time that a statically-determined characteristic of the program holds

```
namespace std {
  template<intmax_t _Num, intmax_t _Den = 1>
  struct ratio
  {
    static_assert(_Den != 0, "denominator cannot be zero");
    static_assert(_Num >= -__INTMAX_MAX__ && _Den >= -__INTMAX_MAX__, "out of range");
    …
  };
}
```

Error management in practice

- assert
- **Class invariant -** a condition established by a constructor
- A member function
  - can assume the class invariant at function entry
  - can make it invalid while performing work
  - must re-establish it at the end

```
template<class T> class vector {
  T* start_;
  T* finish_;
  T* end_of_storage_;

  vector(…) {
    …
    assert(start_ <= finish_ && finish_ <= end_of_storage_);
  }
  void push_back(…) {
    assert(start_ <= finish_ && finish_ <= end_of_storage_);
    …
    assert(start_ <= finish_ && finish_ <= end_of_storage_);
  }
};
```

Error management in practice

- assert
- **Loop invariant -** a condition valid before and after each iteration of a loop
  - but not necessarily in the middle of an iteration

```
buffer::iterator m_buffer_current = m_buffer.begin();

// establish the invariant
ssize_t n_written = std::distance(m_buffer.begin(), m_buffer_current);
assert(n_written == std::distance(m_buffer.begin(), m_buffer_current);

while (m_buffer_current != m_buffer.end()) {
  …
  n_written = …;
  m_buffer_current = …;
  assert(n_written == std::distance(m_buffer.begin(), m_buffer_current);
}

assert(m_buffer_current == m_buffer.end()); // ! loop condition
assert(n_written == std::distance(m_buffer.begin(), m_buffer.end());
```

Error management in practice

- assert
- **Function pre-condition** – contraints imposed to the arguments used to call the function

```
void string::insert(iterator p, char c)
{
  assert(p >= begin() && p <= end());
  …
}
```

- Do not use exceptions here, it's a useless performance penalty

Error management in practice

- assert
- Logical **checkpoint** in the code

```
void print_node_with_role(Nodes const& nodes, Role role)
{
  auto node_it = std::find_if(
      std::begin(nodes),
      std::end(nodes),
      [=](Node const& node) { return node.role() == role; }
  );
  assert(node_it != std::end(nodes)); // a node with that role must be there
  std::cout << *node_it;
);
```

- NB: conditions can be expensive or even impossible to check
  - consider compiling with `-DNDEBUG` to disables asserts in production

Error management in practice

- Exceptions
- Failure to establish the invariant in the constructor

```
class NeverNullString {
  char* s_; // s_ != nullptr && s_ is null-terminated (invariant)
 public:
  NeverNullString() = delete;
  NeverNullString(char const* s) {
    assert(s != nullptr); // && s is null-terminated (pre-condition)
    try {
      s_ = new char[strlen(s) + 1];
    } catch (std::bad_alloc& e) {
      throw CannotCreateString{};
    }
    strcpy(s_, s,);
  }
  …
};
```

- At the end of a successful execution of the constructor
  - `s_` is not `nullptr`, otherwise `CannotCreateString` is thrown
  - If the pre-condition is true, `strcpy` terminates `s_` with `\0`

Contract-based programming

- Summary
  - use static_assert and assert to enforce logical correctness in strategic places in code
    - class invariant
    - loop invariant
    - function pre-condition
    - checkpoint
  - use an exception to communicate failure to satisfy the post-condition of a function
    - tipically related to the acquisition of a resource

Hands-on

- Identify invariants, pre-conditions, post-conditions and other logical checkpoints in the `String` implementation and try to express them with `static_asserts`, `asserts` and exceptions

Memory model

- With C++11, finally C++ has a memory model that contemplates a multi-threaded execution of a program
- A thread is a single flow of control within a program
  - Every thread can potentially access every object and function in the program
  - The interleaving of each thread's instructions is undefined

Thread 1	Thread 2
`y = 1;` `r1 = x;`	`x = 1;` `r2 = y;`

Execution 1	Execution 2	Execution 3

`y = 1;`
`r1 = x;`
`x = 1;`
`r2 = y;`
`// r1 == 0, r2 == 1`
	`x = 1;`
	`r2 = y;`
	`y = 1;`
	`r1 = x;`
	`// r1 == 1, r2 == 0`
		`y = 1;`
		`x = 1;`
		`r2 = y;`
		`r1 = x;`
		`// r1 == 1, r2 == 1`

Memory model

- C++ guarantees that two threads can update and access **separate** memory locations without interfering with each other
- For all other situations updates and accesses have to be properly synchronized
  - atomics, locks, memory fences
- If updates and accesses to the same location by multiple threads are not properly synchronized, there is a **data race**
  - undefined behavior
- Data races can be made visible by transformations applied by the compiler or by the processor for performance reasons