Efficient C++ Coding

From Pointers to Values

Francesco Giacomini – INFN-CNAF

ESC'14, 2013-10-21

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

Hands-on instructions

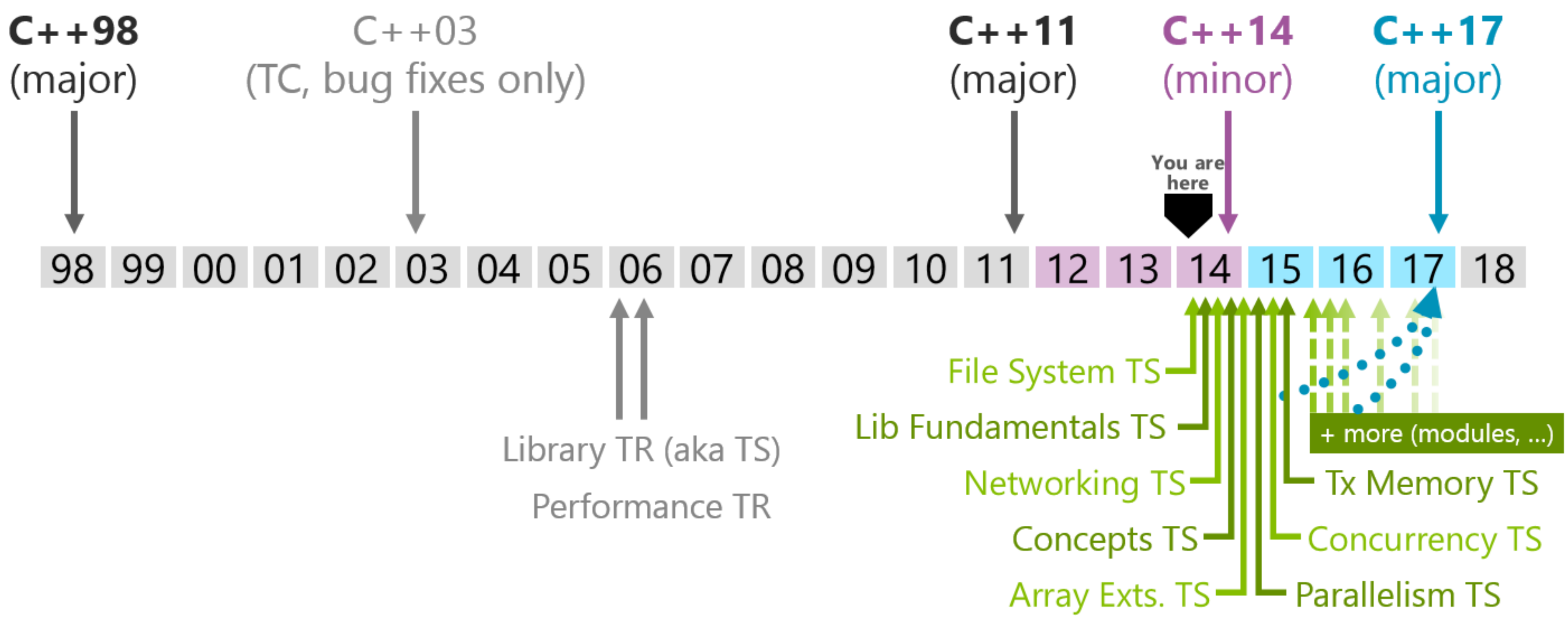
- gcc49env
  - compile with `-std=c++11` or `-std=c++1y`
- To check generated assembler with different compilers
  http://gcc.godbolt.org
- To compile and run with different compilers
  http://coliru.stacked-crooked.com/
- C++ reference
  http://en.cppreference.com/

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);
}
```

```
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)
```

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

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) { return abs(l) < abs(r); }); // C++11
```

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
```

Hands-on

- Fill a vector with random integers between 0 and N
  - std::random
  - std::generate_n
- Sort the vector
  - in ascending order
  - in descending order
- Erase from the vector the multiples of 3 or 7
  - std::remove_if
  - vector::erase

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()
```

pimpl idiom

- Pointer to implementation

```
// in E.hh

class E {
  class Impl;
  Impl* impl_;
public:
  E();
  ~E();
  // …
};
```

```
// in E.cc

class E::Impl {
  // actual data members
}

E::E(): impl_(new Impl) {
  // …
}

E::~E() {
  delete 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
  - 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(…);
    }
    ```
    - note also the use of a data member for a variable that could have been local to the function
  - ```
    TH1F *hfix = new TH1F("hfix","hfix title",nbins,xlow,xup);
    ```
    (the guide doesn't even mention the delete)
    ```
    TH1F hfix{"hfix","hfix title",nbins,xlow,xup};
    ```

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

```
{
  TH1F *hfix = new TH1F("hfix","hfix title",nbins,xlow,xup);
  // …
} // ops, forgot to delete hfix

{
  TH1F *hfix = new TH1F("hfix","hfix title",nbins,xlow,xup);
  // …
  delete hfix;
  // …
  delete hfix; // 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

```
{
  TH1F *hfix = new TH1F("hfix","hfix title",nbins,xlow,xup);
  // potentially-throwing code
  delete hfix; // 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

Example: pimpl

```cpp
// in E.hh

class E {
  class Impl;
  using ImplP = std::shared_ptr<Impl>;
  ImplP impl_;
public:
  E(…);
  // …
};
```

```cpp
// in E.cc

class E::Impl {
  // actual data members
}

E::E(…): impl_{make_shared<Impl>(… )} {
  // …
}
```

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

Hands-on

- Consider the following class interface

```
class DirectoryReader
{
  // …
 public:
  DirectoryReader(std::string const& name);
  ~DirectoryReader();
  std::vector<std::string> entries() const;
  std::string name() const;
};
```

1. Implement it without the pimpl idiom
2. Implement it with the pimpl idiom
    1. Use std::shared_ptr<> for the pimpl
    2. Use std::unique_ptr<> for the pimpl
3. Implement the copy constructor and the copy assigment

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

Returning 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
```

  - or, use “out” parameters

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

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

Returning a value from a function

- Wouldn't it be better to write

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

auto lo = make_large_object(); // is there a copy here?
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 ( … );
}

Type t = make_urvo( … );
```

- Named

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

Type 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_without_rvo( … )
{
  Type result;
  if ( … ) {
    return Type( … );
  }
  return result;
}

Type t = make_without_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 objects are called rvalues
  - and you can't take their address

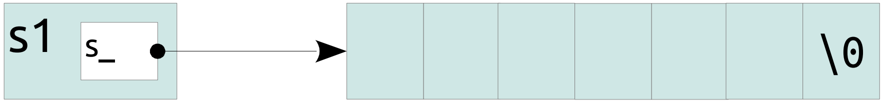
Copy vs move

```
String s1("...");
```

Copy vs move

```
class String {
  char* s_; // 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_); }
};
```

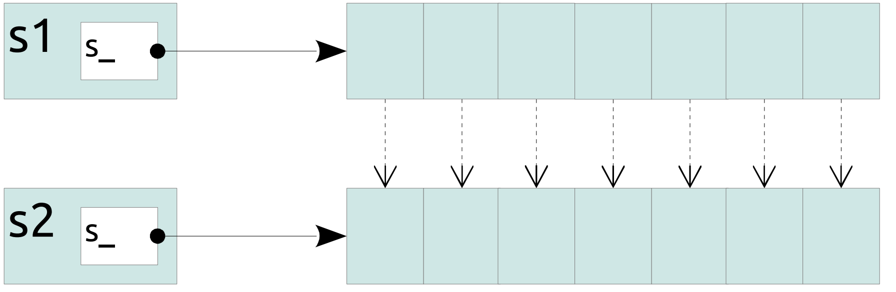
```
String s1("...");
```



Copy vs move

```
class String {
  char* s_; // 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_); }
};
```

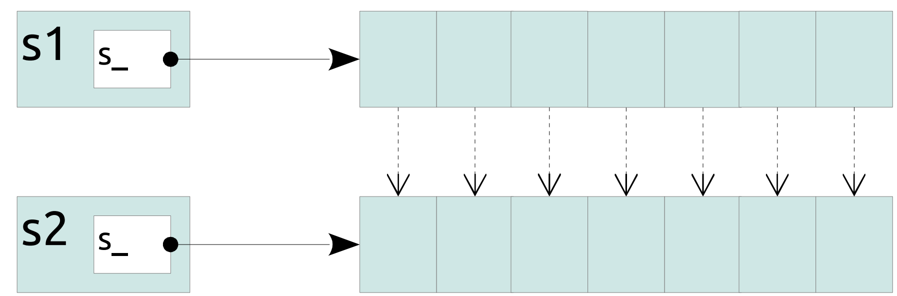
```
String s1("...");
String s2(s1);
```



Copy vs move

```
class String {
  char* s_; // 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("...");
String s2(s1);
```

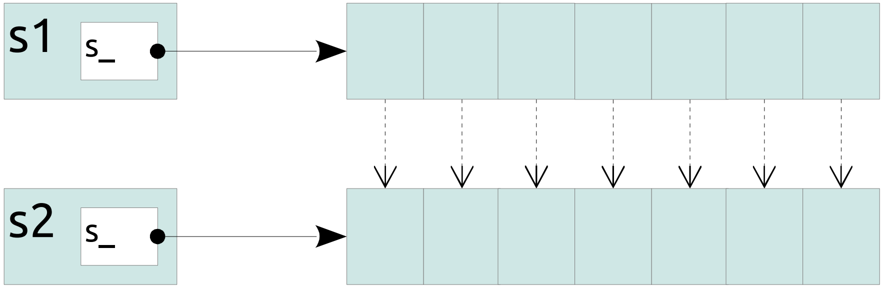



```
String s3(get_string());
```

Copy vs move

```
class String {
  char* s_; // 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("...");
String s2(s1);
```

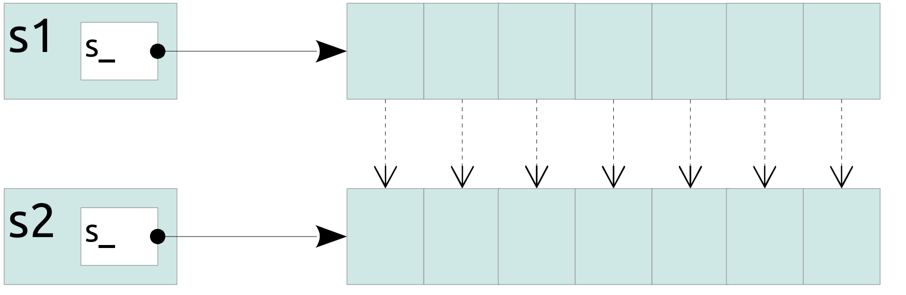



```
String s3(get_string());
```

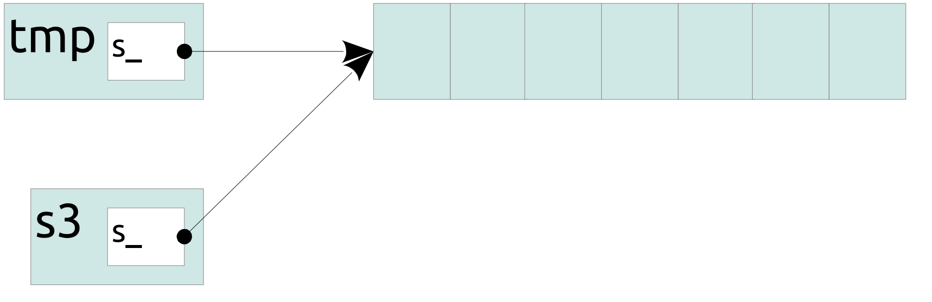
Copy vs move

```
class String {
  char* s_; // 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();
```

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



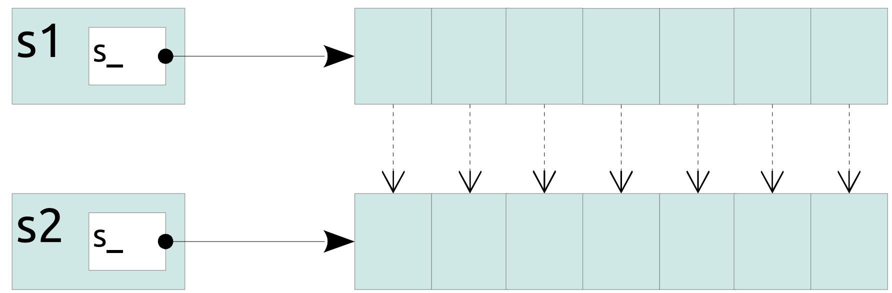

```
String s3(get_string());
```



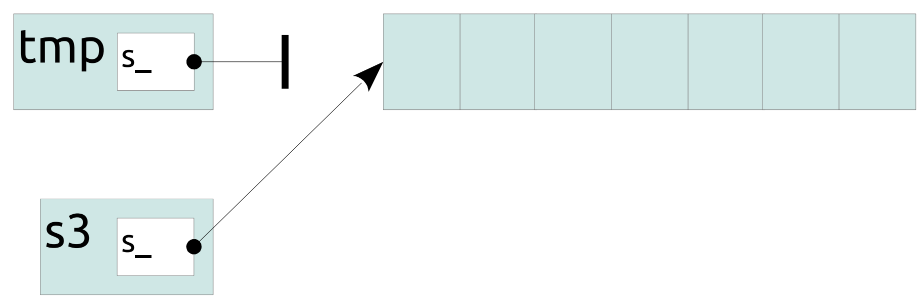
Copy vs move

```
class String {
  char* s_; // 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();
```

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




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

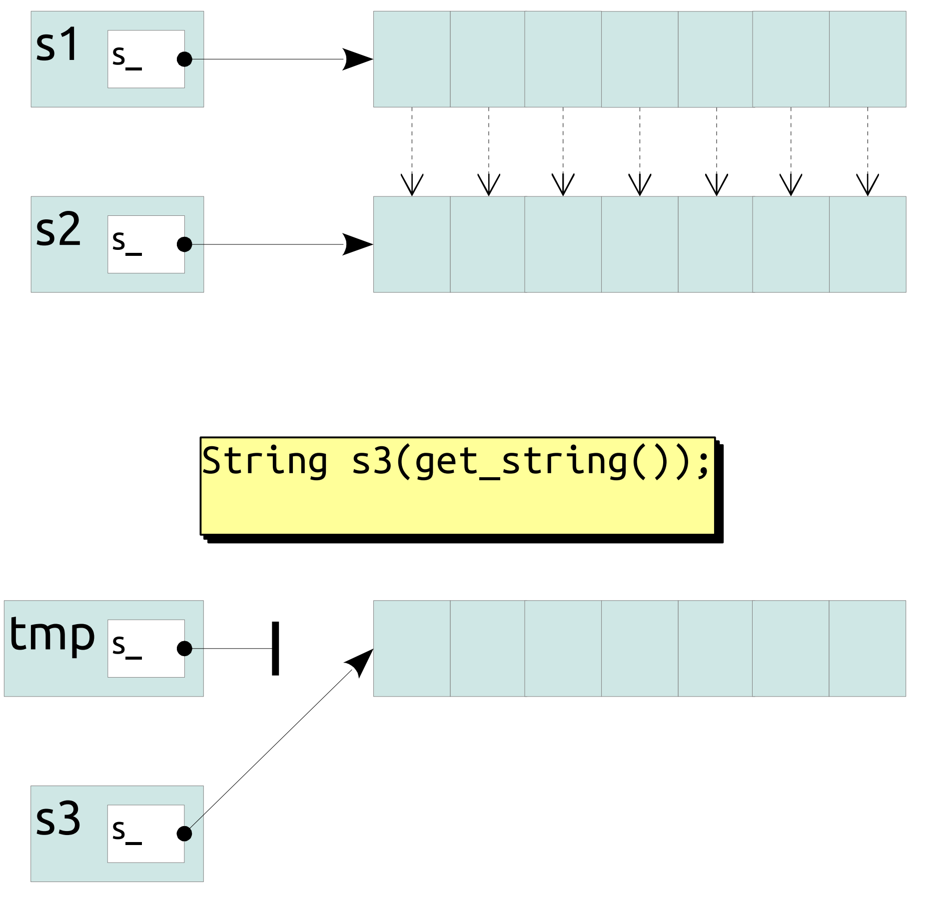


Copy vs move

```
class String {
  char* s_; // 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();
```

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

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



rvalue references

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

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

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

lvalues can be explicitly transformed into rvalues

```
void foo(String&&);
void foo(String const&);

foo(s);            // calls foo(String const&)
foo(std::move(s)); // calls foo(String&&); I don't care about s anymore
s.size();          // undefined
```

Special functions

- A class in C++11 has now 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 // non-copyable, movable
{
  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

```
// a non-copyable class in C++98
struct C
{
  C() {}
private:
  C(C const&);
  C& operator=(C const&);
};

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

declare, but don't define, the copy constructor and the copy assignment operator as private

```
// a non-copyable class in C++11
struct C
{
  C() {}
  C(C const&) = delete;
  C& operator=(C const&) = delete;
};

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

copy constructor and copy assignment operator can even be public

The mechanism is more general: any function can be deleted

```
void f(double);

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

void f(int) = delete;

f(1);  // 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 it 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 it with code

```
struct C
{
  C() = default;
  C(C const&) = default;
  C& operator=(C const&) = default;
};

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

Remember that the default constructor is generated only if no other constructor is present

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

Mechanisms for error management

The sooner errors are identified, the better

- `static_assert`
  - declares a logical assertion that **by design** must be valid at compile time
- `assert`
  - declares a 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
- Return codes
  - C-style
  - 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

```
class X {…};

template<typename T>
class Y {
  static_assert(
    std::is_default_constructible<T>::value,
    "T must be default constructible"
  );
};

Y<X> y; // this would fail if X were not default constructible
```

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

assert

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

```
std::vector<int> make_v_with_at_least_3_items(int n, int v)
{
  assert(n >= 3);
  return std::vector<int>(n, v);
}

int main()
{
  auto v1 = make_v_with_at_least_3_items(4, 1); // ok
  auto v2 = make_v_with_at_least_3_items(2, 1); // abort
}
```

- Useful for 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 separating application logic from error management
- An exception is raised using throw and intercepted with catch

```
struct E {…};
struct 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 g() {
  try {
    // …
    auto s = make_s();
    f();
    // …
  } 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 `terminate()`'d
- During stack unwinding all object destructors are called

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

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

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

Exception specification

- C++98 allowed the possibility to list the exceptions that a function could throw

```
void f() throw(std::runtime_exception, my_exception);
```

- Dynamic exception specifications are deprecated in C++11
  - still supported, but can be removed from a future revision of the standard
  - they were practically useless, if not harmful, anyway
- A `noexcept` specification has been introduced

```
void f() noexcept;
```

- `noexcept` tells the compiler that a function
  - doesn't throw, or
  - is not able to manage possible exceptions → better to terminate
- `noexcept` can depend on a constant expression, to make the function conditionally non-throwing

Exception safety

- Different levels of safety guarantees (for member functions):
  - **Basic** - if an exception is thrown, the object’s invariant is still valid and no resource is leaked
    - the object is usable (at least destroyable) although the contents may be unspecified
    - every class should provide at least the basic guarantee
  - **Strong** - if an exception is thrown, the object's state is as it was before the function was called
  - **No-throw** - no exception leaves the function
    - the function should be marked as `noexcept`

Destructor and noexcept

- In C++11 the destructor is by default noexcept
  - i.e. releasing a resource cannot fail
  - technically an incompatibility wrt to C++98

```
struct X {
  ~X() { throw 1; }
};

int main()
{
  try {
    X x;
  } catch (...) {
      std::cout << "exception\n";
  }
}
```

- The exception is caught in C++98 (“exception” is printed)
- The program is terminate'd in C++11

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

Passing parameters to a function

- By (smart) *, by &, by &&, by value?
- C++98's default advice still makes sense
  - pass primitive and small types by value
  - pass large types by (const)&
- Don't pass by smart pointer, unless the smart pointer itself is needed in the callee

```
void use_pointer(shared_ptr<Type> p) {
  p.reset( … );
}

void use_type(Type& t) {
  t. … ;
}

shared_ptr<Type> p = std::make_shared<Type>( … );
use_pointer(p);
use_type(*p);
```

Passing parameters to a function

- C++11 allows to add a function overload 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);
```

The importance of being noexcept

- Consider

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

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

- How much does noexcept affect the performance?

The importance of being noexcept

- Consider

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

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

- How much does noexcept affect the performance?
  - without noexcept: 1077 ms
  - with noexcept: 66 ms
- If the move can throw, vector will copy, not move, data

Move and noexcept

- If move operations are noexcept the compiler can apply significant optimizations
  - aim at it
- `T::operator=(T&&)` is typically easy to make noexcept
  - if swap is noexcept (which it should be), just swap all data members
- `T& T::T(T&& tmp)` may be more difficult
  - start with one object (`tmp`), end up with two (`*this` and `tmp`)
  - probably `T::T()` must be `noexcept` as well
    - which is not obvious if a resource has to be acquired

Error management in practice

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

```
#include <type_traits>

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

static_assert(
  std::is_nothrow_move_constructible<String>::value,
  "String must be nothrow move constructible"
)
```

- String is nothrow move constructible

Error management in practice

- assert
- The class invariant must be true at the beginning of a member function
- A pre-condition must be true at the beginning of a function

```
class String
{
  size_t size() const {
    assert(s_ != nullptr);        // && s_ is null-terminated (invariant)
    return strlen(s_);
  }
  char operator[](size_t i) const {
    assert(s_ != nullptr);        // && s_ is null-terminated (invariant)
    assert(0 <= i && i < size()); // i is within the valid range (pre-condition)
    return s_[i];
  }
};
```

- Conditions can be expensive or even impossible to check
  - -DNDEBUG

Error management in practice

- assert
- Check that a certain logical condition must be true during the execution

```
std::vector<X> v { … };

X& search_in_v(std::string const& name) {
  auto it = std::find_if(
    begin(v),
    end(v),
    [](X const& x) { return x.name() == name; }
  );
  assert(it != end(v));
  return *it;
}
```

(Assuming that **by design** an X with that name must be there)

Error management in practice

- Exception
- Failure to establish the invariant in the constructor

```
class String {
  char* s_; // s_ != nullptr && s_ is null-terminated (invariant)
 public:
  String(char const* s) {
    assert(s != nullptr); // && s is null-terminated (pre-condition)
    size_t size = strlen(s) + 1;
    s_ = new char[size];
    memcpy(s_, s, size);
  }
  // …
};
```

- At the end of a successful execution of the constructor
  - `s_` is not `nullptr`, otherwise new would have thrown `bad_alloc`
  - **If** the pre-condition is true, `memcpy` copies also a `'\0'` at the end of `s_`
- `bad_alloc` means failure to establish the invariant

Strong exception safety

```
class String {
  char* s_;
public:
 String(String const& other) {
    size_t size = strlen(other.s_) + 1;
    s_ = new char[size];
    memcpy(s_, other.s_, size);
  }
  String& operator=(String const& other)
  {
    String tmp(other);
    std::swap(s_, tmp.s_);
    return *this;
  }
  // …
};
```

- How to implement the copy assignment, guaranteeing strong exception safety
  - if the operation fails the object state remains unaltered
- Copy & swap
  - acquire the needed resources creating a temporary copy of the source
    - can throw
  - swap the resources of the temporary copy with the resources of `*this`
  - the destructor of the temporary copy releases the old resources of `*this`
  - swap is noexcept

De-virtualization

- Virtual functions have a cost
  - indirect call through a pointer kept in the virtual table
- If the compiler can determine the exact dynamic type of an object, it can call the function directly
  - and possibly inline it

```
struct B {
  virtual int fun() { return 31; }
};
struct D: B {
  int fun() { return 42; }
};
struct E: D {
};

int main()
{
  B* b = new D;
  b->fun();
}
```

here the compiler can easily see that b is actually a D and call directly, and inline, D::fun()

De-virtualization

- The compiler however may not see enough code or not be able to prove what the dynamic type of an object is
- The compiler can be helped telling it either
  - that a virtual function cannot be overridden any more, or
  - that an entire class cannot be derived from

here the compiler can prove that d->fun() cannot be overridden and statically determine to call, and inline, D1::fun()

here the compiler can prove that d->fun() cannot be overridden and statically determine to call, and inline, B::fun()

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

int f(D1* d) { return d->f(); }
int f(D2* d) { return d->f(); }
```

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)
```

Memory model

- 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

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

→

Thread 1	Thread 2
`r1 = x;`	`r2 = y;`
`y = 1;`	`x = 1;`
`// r1 == 0`	`// r2 == 0`

Take-away messages

- Strive not to use pointers
  - if you have to, manage memory with smart pointers
- When designing a class define its invariant, pre-conditions and post-conditions
  - check invariants and pre-conditions with (static) asserts
  - use exceptions to communicate failure to (re-)establish invariants or to satisfy post-conditions
- Help the compiler to enable the return value optimization
- Strive to make at least the move operations noexcept
- Make your code const-correct
- Consider static polymorphism instead of dynamic polymorphism

Further reading

- News, Status & Discussion about Standard C+
  http://www.isocpp.org
- The C++ Standards Committee
  http://www.open-std.org/jtc1/sc22/wg21/
- C++ Now Conference
  http://cppnow.org/
- The C++ Conference
  http://cppcon.org/
- boost C++ libraries
  http://www.boost.org/

Hands-on

- Complete the String implementation so that:
  - it is default constructible
  - it is constructible from a C string
  - it is movable and copyable
  - it provides a size() member function
  - it provides access to single characters with operator[]
  - it provides a `c_str()` member function
  - it provides support for iterators
- Put the noexcept declaration where applicable
- Be generous with `static_assert` and `assert`
- Experiment with alternative internal representations