Efficient C++ Coding

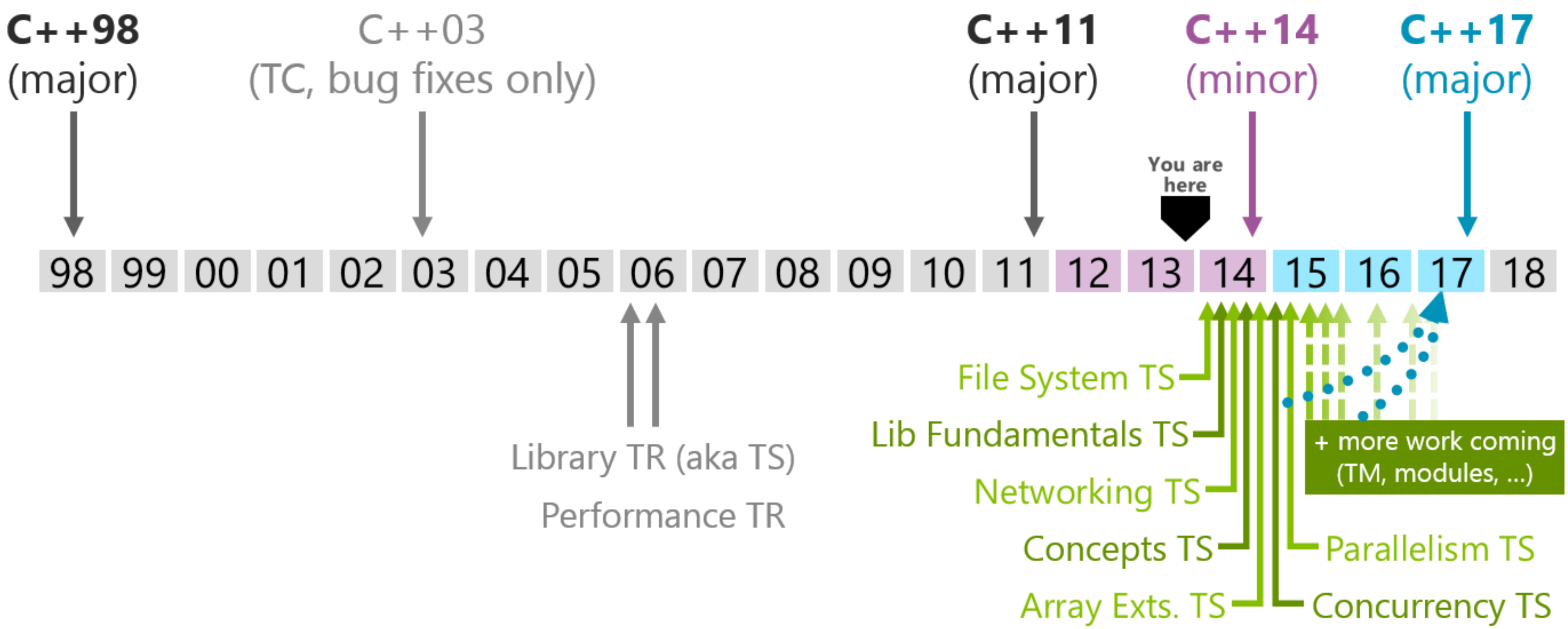
How modern C++ improves your code

Francesco Giacomini – INFN-CNAF

ESC'13, 2013-10-22

A bit of C++ history

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



Sources about C++11

- The standard
  - a document close to the final text is freely available at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf
- Official documents available at the C++ standard committee's site http://www.open-std.org/jtc1/sc22/wg21/docs/papers/
- The C++11 FAQ by B. Stroustrup http://www.stroustrup.com/C++11FAQ.html
- The boost libraries http://www.boost.org/
- GoingNative http://channel9.msdn.com/Events/GoingNative/2013

- For an introduction to C++98

  https://www.cnaf.infn.it/~giaco/c++/introduction_to_c++.pdf

Efficiency

- For the computer → performance
- For the developer → productivity
- For the reader → maintainability

The Core Language

To the Standard Library

>>

```
#include <vector>
#include <complex>

std::vector<std::complex<double>> v;  // error in C++98, ok in C++11

std::vector<std::complex<double> > v; // ok in both C++98 and C++11
```

Note the space

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

Initializer lists

- Implemented with std::initializer_list<T>
  - E.g. {1, 2, 3, 4, 5} has type std::initializer_list<int>
  - Consider it a read-only container
    - Availability of size(), begin(), end(), iterators, etc.
  - Elements must be uniform

```
void f(std::initializer_list<int> l)
{
  for (std::initializer_list<int>::const_iterator b = begin(l), e = end(l); b != e; ++b) {
    std::cout << *b << '\n';
  }
}
template<typename T>
class vector
{
  vector(std::initializer_list<T> l) { … };
};
```

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
```

auto

```
std::map<std::string, int> m;

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

Why should I tell the compiler what the type of iter 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

```
auto a;                  // error, no initializer
auto i = 0;              // i has type int
auto d = 0.;             // d has type double
auto const e = 0.;       // e has type double const
auto const& g = 0.;      // g has type double const&
auto f = 0.f;            // f has type float
auto c = "ciao";         // c has type char const*
auto p = new auto(1);    // p has type int*
auto int i;              // OK in C++98, error in C++11
```

```
template<typename T, typename U>
void op(T t, U u)
{
  …
  ??? s = t * u; // what's the type of s?
  …
}
```

```
template<typename T, typename U>
void op(T t, U u)
{
  …
  auto s = t * u;
  …
}
```

range for

```
std::vector<Account> v;

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

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

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

```
std::vector<Account> v;

for (Account a: v) { // or, rather, Account const&
  print(a);
}
for (auto u: v) { // or, rather, auto const&
  print(u);
}
for (auto& u: v) {
  update(u);
}
```

range for

- A range can be:
  - an array
    - e.g. int a[10];
  - a class C 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);
```

nullptr

- nullptr is a literal denoting the null pointer

```
f(int);
f(int*);

f(0);       // calls f(int)
f(nullptr); // calls f(int*)

int* p = nullptr;
int* q = 0; // still valid and p == q
```

- nullptr is not an integer

```
int i = nullptr; // error
```

- NULL was **not** a solution

```
// stddef.h
#ifdef (__cplusplus)
#define NULL 0
```

In-class member initializers

Allow to specify the initialization expression of a non-static data member when it is declared

```
struct C
{
  int i_;
  C(): i_(5) {}
  C(int i): i_(i) {}
};

C c1;    // c1.i_ == 5
C c2(3); // c2.i_ == 3
```

```
struct C
{
  int i_ = 5;
  C() {}
  C(int i): i_(i) {}
};

C c1;    // c1.i_ == 5
C c2(3); // c2.i_ == 3
```

the initialization in the constructor takes precedence

Simplifies the initialization of objects, especially in presence of multiple constructors, that need to be kept in sync

In-class member initializers

Beware side effects in the initializer

```
double global_d;

std::string get_string() { return "hello, world!"; }
double get_double() { global_d = 3.; return 1.;}

class A
{
  int i_ = 5;
  std::string s_ = get_string();
  double d_ = get_double(); // ignored!
public:
  A(): d_(2.) {}
};

A a;
// a.i_ == 5, a.s_ == "hello, world!", a.d_ == 2., global_d == 0
```

side effect

Delegating constructors

A constructor can delegate to another constructor

```
class C
{
  std::string k_;
  int v_;
  void init(std::string const& k, int v)
  {
    k_ = k;
    v_ = v;
    // validate k_ and v_
  }
public:
  C(std::string const& k, int v) { init(k, v); }
  C(int v) { init("default", v); }
};
```

```
class C
{
  std::string k_;
  int v_;
public:
  C(std::string const& k, int v)
    : k_(k), v_(v)
  {
    // validate k_ and v_
  }
  C(int v): C("default", v) {}
};
```

- No need of an auxiliary function
- Favor initialization over assignment

Inheriting constructors

Re-use constructors defined in a base class

```
struct B
{
  void f(int);
  void f(double);
  B(int);
  B(double);
};

struct D1: B             // C++98
{
  using B::f;
  void f(int);           // prefer this to B::f(int)

  D1(int);
  D1(double d): B(d) {} // forward to B::B(double)
};

struct D2: B             // C++11
{
  using B::B;            // re-use B's constructors
  D2(int);               // prefer this to B::B(int)
};
```

Deleted functions

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
```

Defaulted functions

- 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

long long

- A longer integer, at least 64 bits
  - signed and unsigned

```
long long i1 = 4611686018427387903;
auto i2 = 1LL;          // i2 has type long long
unsigned long long i3 = 4611686018427387903;
auto i4 = 1ULL;         // i4 has type unsigned long long
```

Exception specification

- Dynamic exception specifications are deprecated
  - still supported, but can be removed from a future revision of the standard

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

  - 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

Raw string literals

- A string literal that doesn't recognize C++ escape sequences (e.g. '\n', '\\', '\"', ...)
- Address readability difficulties introduced, e.g., with regular expressions and XML text

```
cout << "\\\\\""; // prints \\" (3 characters)
cout << R"(\\")"; // idem, but easier

cout << "\n";     // prints a newline
cout << R"(\n)";  // prints \n (2 characters)

// how to print )" (2 characters)
cout << R"()")";     // error
cout << R"*+()")*+"; // ok
```

an (almost) arbitrary sequence of characters

User-defined literals

C++98 foresees literals only for built-in types

```
123     // int
123U    // unsigned int
123L    // long int
12.3    // double
12.3F   // float
12.3L   // long double
'h'     // char
"hello" // char[6]
```

C++11 supports literals for user-defined types as well

```
// for example
123s       // seconds
12us       // microseconds
42.1km     // kilometers
23.3m2     // squared meters
"hi"s      // as an std::string
.5i        // imaginary
10101000b // binary

Speed v1 = 9.8m / 1s;     // ok
Speed v2 = 100m / 1s2;    // error
Speed v3 = 100 / 1s;      // error
Acceleration a = v1 / 1s; // ok
```

Improve type safety, with no runtime overhead

User-defined literals

UD literals are mapped to the corresponding types via literal operators

```
std::complex<double> operator "" i(long double d)
{
  return {0, d};
}

std::string operator "" s(char const* chars, size_t size)
{
  return std::string(chars, size);
}
```

- Certain suffixes (those not starting with _) are actually reserved for future standardization
- Namespaces help prevent suffix clashes

enum class

- `enum` in C++98 has several issues:
  - absence of strong scoping
  - implicit conversion to integer
  - inability to specify the underlying type

```
enum E1 { red };
enum E2 { red }; // error
```

```
enum Color { red, green, blue };
int c = red; // ok
void f(int);
f(green);    // ok
```

```
sizeof(Color); // 4 (!) bytes on my machine
```

enum class

- `enum class` in C++11 addresses the `enum` issues:
  - strong scoping

```
enum class E1 { red };
enum class E2 { red }; // ok
```

  - no implicit conversion to integer

```
enum class Color { red, green, blue };

int c = red;      // error

void f(int);
f(green);         // error
f(Color::green); // error

void g(Color);
g(green);         // error
g(Color::green); // ok
```

  - ability to specify the underlying type

```
enum class Color: char { red, green, blue };
sizeof(Color);   // 1 (on any machine!)
```

Static assertions

Check that a certain constant boolean expression is satisfied during compilation, otherwise fail with the specified message

```
static_assert(sizeof(long) >= 8, "64-bit code generation required for this library.");
```

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

constexpr

- Generalize the concept of constant expressions, extending it to include
  - calls to “sufficiently simple” functions
    - more flexibility in C++14
  - objects of user-defined types constructed from “sufficiently simple” constructors
- A way to guarantee that an initialization is computed at compile time
  - favor const-correctness
  - performance benefit
- `constexpr` functions can be called also at run-time

factorial at compile time

```
// general case
template<int N>
struct Factorial // C++98
{
  static const int value = N * Factorial<N-1>::value;
};

// basic case
template<>
struct Factorial<1>
{
  static const int value = 1;
};

constexpr int factorial(int N) // C++11
{
  return N > 1 ? N * factorial(N - 1) : 1;
}

constexpr int f1 = Factorial<10>::value;
constexpr int f2 = factorial(10);
static_assert(f1 == f2, "Factorial<10>::value == factorial(10)");
```

constexpr

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

constexpr Complex c {12.3, 4.56}; // ok, initialized at compile time
constexpr double r = c.real;      // ok, initialized at compile time

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

void f(Complex c)
{
  constexpr double r1 = c.real; // error, can't evaluate at compile time
  double r2 = c.real;           // ok
  constexpr Complex c1 = 2i;    // ok
}
```

constexpr implies const

Lambda expressions

- 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
```

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

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

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

Lambda expressions

- 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
```

Copy vs move

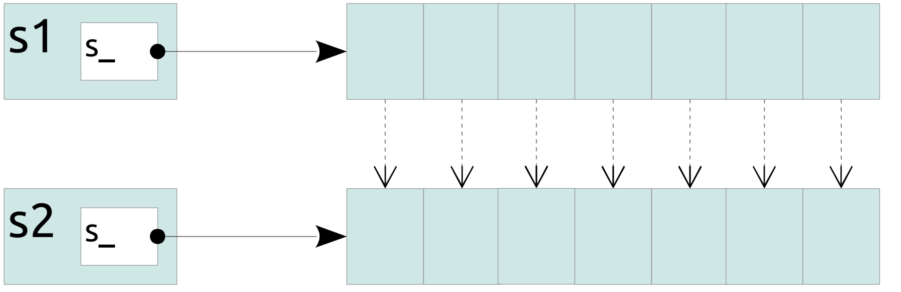
```
String s1("...");
String s2(s1);          // (1)

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

- In (1) the deep copy is necessary, because s1 still esists 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

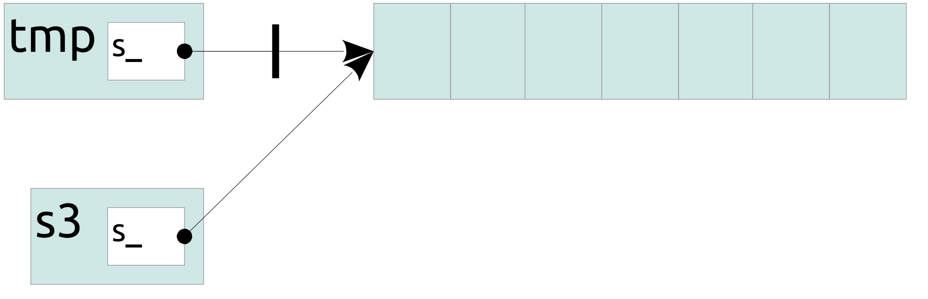
Copy vs move

```
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(??? other): s_(other.s_) {
    other.s_ = nullptr;
  }
};

String get_string();
```

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




```
String s1, s2;
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&& other)
    : s_(other.s_)
  {
    other.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

```
String s2(std::move(s1)); // I don't care about s1 anymore
s1.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
};
```

- 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;
};
```

Move in the standard lib

```
namespace std
{
  template<typename T, …>
  class vector
  {
  public:
    vector(vector&& other);
    vector& operator=(vector&& other);
    void push_back(T&& value);
    iterator insert(iterator position, T&& value);
    …
  };
}

vector<string> read_from_file(std::string const& filename);
string get_another_string();

vector<string> v(read_from_file("strings.txt"));
v.push_back(get_another_string());
```

move semantics applied to the container

move semantics applied to the elements

steal from the temporary returned by the function

Reminder on exception safety

- Different levels of exception safety guarantees:
  - **No guarantee** – if an exception occurs, anything can happen
  - **Basic** – if an exception occurs, at least object invariants are preserved → no resources are leaked, the object is correctly destructible, etc., but the original object value may not be preserved
  - **Strong** – if an exception occurs, the original object value is preserved
  - **No throw** – if an exception occurs, it is swallowed and managed directly by the implementation, without any visible effect
- In general the higher the level of safety, the higher the overhead
  - strive to provide at least the basic guarantee
  - the no throw guarantee (→ noexcept) enables significant optimizations, e.g. in the standard library

Example for copy assignment

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

- The copy constructor can throw because new can throw
  - if the memory allocation fails
- The copy assignment can throw because the copy constructor can throw
  - but if it throws the original value is preserved → **strong** exception safety guarantee
- The move assignment provides the **no throw** guarantee → marked as noexcept
- Also the move constructor can be marked noexcept
- Note that the move assignment is not protected against self-assignment
  - s = std::move(s)
  - but at least it doesn't crash

Passing by value

- Perceived as inefficient → pass by const&
- In fact compilers are allowed to do *copy elision* in some circumstances
  - they are good at it and getting better
- General advice: pass sink arguments by value and move them into destination
  - i.e. arguments that would in any case be copied within the function body

```
class runtime_error
{
  String msg_;
public:
  runtime_error(String msg)
  : msg_(std::move(msg))
  {
  }
};
```

```
String s;
// pass an lvalue
runtime_error e1(s);
// pass a temporary
runtime_error e2(get_string());
```

	by const& + copy	by value + move
pass lvalue	1 copy	1 copy + 1 move
pass rvalue	1 copy	1 move

For a POD: move == copy

Return value optimization

- Copy elision applies also to return values
  - (U)RVO → (Unnamed) Return Value Optimization
  - NRVO → Named Return Value Optimization
- If the compiler is not able to elide the copy, it tries first to *move* the return value into destination and only copies it as a last resort

```
String urvo()
{
  return {}; // returns an unnamed default-constructed String
}

String nrvo()
{
  String result;
  return result; // returns a named default-constructed String
}

String s1 = urvo();
String s2 = nrvo();
```

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 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

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

// in another translation unit

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

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()

De-virtualization

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

```
struct B {
  virtual int f() = 0;
};
struct D1: B {
  int f() { 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)
```

Trailing return type

- That auto specifier in place of the return type of a function means that the declarator of that function shall include a trailing return type

what's the type of the returned value?

```
template<typename T, typename U>
??? op(T t, U u)
{
  return t * u;
}
```

```
template<typename T, typename U>
auto op(T t, U u) -> decltype(t * u)
{
  return t * u;
}
```

auto here means “to be specified later”
t and u are not known until after their declaration

- This form can be used for any function

```
auto op(double d, double e) -> double
{
  return d * e;
}
```

Type aliases

- A more powerful typedef
  - enable template typedef

```
// can be used in place of the usual typedef
typedef vector<Employee> Employees;
using Employees = vector<Employee>; // equivalent, probably preferable
Employees employees;

template <class T, size_t N> class array;           // C++11 fixed-size array
array<int, 100> a1;                                 // an array of 100 ints

template<class T>  using array_100 = array<T, 100>; // N is bound to 100
array_100<int> a2;                                  // an array of 100 ints

template<size_t N> using int_array = array<int, N>; // T is bound to int
int_array<100> a3;                                  // an array of 100 ints

static_assert(is_same<array<int, 100>, array_100<int>>::value, "different types");
static_assert(is_same<array<int, 100>, int_array<100>>::value, "different types");
```

Variadic templates

- Provide templates with the ability to accept any number of template arguments
  - How to implement a type-safe printf?

```
printf("Hello, %s.\n", "Francesco");                         // 2 args
printf("The value of %s is %g.\n", std::string("pi"), 3.14); // 3 args
```

  - How to build a tuple?
    - A tuple is a heterogeneous, fixed-size collection of values

```
std::tuple<int, Employee, string> r {
    12345, Employee("Giacomini"), string("INFN")
}; // 3 elements

std::tuple<Point, Length, Color, Thickness> l {
    Point(1.,2.), 12mm, Red, .1in
}; // 4 elements
```

Variadic templates

Use of recursion: general case + one or more basic cases

```
// general case
template<typename T, typename... R>        Template type parameter pack
void print(T value, R... rest)       Function parameter pack
{
  cout << value << ' ';
  print(rest...);      // Recursive call
}              Parameter expansion

// basic case
template<typename T>
void print(T t)
{
  cout << t;
}

print(1, 2.9, "hello", '\n'); // T = int, R = {double, char const*, char}
print(2.9, "hello", '\n');    // T = double, R = {char const*, char}
print('\n');                  // basic case, T = char
```

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

→

Thread 1	Thread 2

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

The Standard Library

To the Core Language

Overview of changes

- Adoption of new language features in the library
  - examples
- New data structures
- New algorithms
- New libraries

Use of constexpr

```
namespace std {
  template<> class complex<double>
  {
  public:
    constexpr complex(double re = 0.0, double im = 0.0);
    constexpr double real();
    constexpr double imag();
    …
  };
}

constexpr std::complex c{1., 2.};
constexpr double re = c.real();
constexpr double im = c.imag();
```

= default, = delete

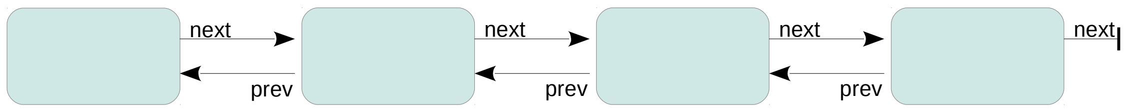
```
namespace std {
  template<typename T> class shared_ptr
  {
    public:
      // copyable, let the compiler implement the functions
      shared_ptr(shared_ptr const&) noexcept = default;
      shared_ptr& operator=(shared_ptr const&) noexcept = default;
    …
  };

  class error_category
  {
  public:
    // not copyable
    error_category(error_category const&) = delete;
    error_category& operator=(error_category const&) = delete;
    …
  };
}
```

Containers

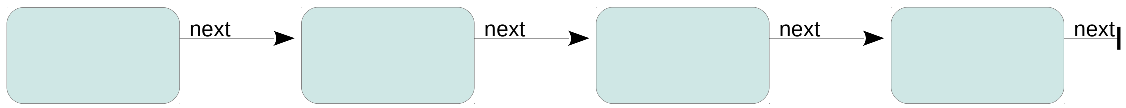
- C++ containers belong to two broad categories
  - sequence: elements are strictly in a linear order and the order is under the control of the developer
    - vector, deque, list, **forward_list**, **array**
  - associative: the order of the elements is under the control of the container itself, based on a key
    - set, multiset, map, multimap, **unordered_set**, **unordered_multiset**, **unordered_map**, **unordered_multimap**

forward_list

- std::list



- std::forward_list
  - more space-efficient, but less functionality



array

- Fixed-size container of homogeneous elements
  - storage is contiguous
  - no space overhead with respect to a C array

```
std::array<int, 4> a {1, 2, 3, 4};
std::array<int, 4> b;                 // {0, 0, 0, 0}
std::array<int, 4> c {1, 2}           // {1, 2, 0, 0}
std::array<int, 4> d {1, 2, 3, 4, 5}; // error
std::array<int> e {1, 2};             // error (size is mandatory)
a[2] = 5;                             // {1, 2, 5, 4}
std::cout << a.size();                // prints 4
auto beg = a.begin();                 // iterator to the first element
auto end = a.end();                   // iterator to one after the last element
int* p = a.data();                    // p points to a[0]
```

Unordered containers

- Also known as hash containers
  - map, multimap, set, multiset
- Similar to their ordered counterparts, but:
  - elements are not ordered
  - no need of a less-than operator being defined
  - need of a hash function, possibly user defined
  - lookup is O(1), instead of O(log N)

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

tuple

- A tuple is an ordered, heterogeneous, fixed-size collection of values
  - a generalization of std::pair
  - an unnamed struct of unnamed members
  - members are accessible with an index

```
std::tuple<Point, Length, Color, Thickness> l {Point(1.,2.), 12mm, Red, .1in};
auto r = make_tuple(12345, Employee("Giacomini"), std::string("INFN"));
// r has type std::tuple<int, Employee, std::string>

Point p = std::get<0>(l);
auto s = std::get<2>(r); // s has type std::string
auto d = std::get<3>(r); // compilation error

int i; Employee e; std::string s;
tie(i, e, s) = r; // i == 12345, e == Employee("Giacomini"), s = std::string("INFN")
```

Algorithms

all_of, **any_of**, **none_of**, for_each, find, find_if, **find_if_not**, find_end, find_first_of, adjacent_find, count, count_if, mismatch, equal, **is_permutation**, search, search_n, copy, **copy_n**, **copy_if**, copy_backward, **move**, **move_backward**, swap, swap_ranges, iter_swap, transform, replace, replace_if, replace_copy, replace_copy_if, fill, fill_n, generate, generate_n, remove, remove_if, remove_copy, remove_copy_if, unique, unique_copy, reverse, reverse_copy, rotate, rotate_copy, random_shuffle, **shuffle**, **is_partitioned**, partition, stable_partition, **partition_copy**, **partition_point**, sort, stable_sort, partial_sort, partial_sort_copy, **is_sorted**, **is_sorted_until**, nth_element, lower_bound, upper_bound, equal_range, binary_search, merge, inplace_merge, includes, set_union, set_intersection, set_difference, set_symmetric_difference, push_heap, pop_heap, make_heap, sort_heap, **is_heap**, **is_heap_until**, min, max, **minmax**, min_element, max_element, **minmax_element**, lexicographical_compare, next_permutation, prev_permutation

Algorithms

```
std::array<int, 6> a {3, 5, 4, 2, 8, 9};
std::all_of(
  begin(a),                             // from here...
  end(a),                               // ... to here
  [] (int i) { return i < 10; }         // if less than 10
); // returns true

std::vector<int> v;
copy_if(
  begin(a),                             // from here...
  end(a),                               // ... to here
  back_inserter(v),                     // push_back to v...
  [] (int i) { return i % 2; }          // ... if odd
); // v == {3, 5, 9}

std::is_sorted(begin(a), end(a)); // returns false
is_sorted(
  begin(v),
  end(v),
  [] (int i, int j) { i % 3 < j % 3; }   // compare mod 3
); // returns false
```

Numeric operations

accumulate, inner_product, partial_sum, adjacent_difference, **iota**

```
std::array<int, 6> a;              // a = {0, 0, 0, 0, 0, 0}
std::iota(begin(a), end(a), 3); // a = {3, 4, 5, 6, 7, 8}
```

Regular expressions

- A form of pattern-matching often used in text processing
  - think grep

beginning of line | 0 or more spaces | 0 or more spaces | 0 or more spaces | 0 or more spaces | end of line

3 letters | 3 letters | 2 digits | 1 letter | 2 digits | 1 letter | 3 digits | 1 letter

```
std::regex e(R"(^[A-Z]{3} *[A-Z]{3} *\d{2}[A-Z]\d{2} *[A-Z]\d{3} *[A-Z]$)");
regex_match("RSS MRA 70A01 H501 S", e); // returns true
regex_match("RSS MRA 70A01 6501 S", e); // returns false
```

```
std::regex e(R"(^([A-Z]{3}) *([A-Z]{3}) *(\d{2}[A-Z]\d{2}) *([A-Z]\d{3}) *([A-Z])$)");
std::string fmt(R"(\1\2\3\4\5)");
regex_replace("RSS MRA 70A01 H501 S", e, fmt); // returns "RSSMRA70A01H501S"
```

Note the use of raw string literals
to avoid excessive escaping of \ characters

Random number generation

- Generators/engines
  - generate uniformly distributed unsigned integers according to specific properties
  - linear congruential, mersenne twister, …
- Distributions
  - generate values that are distributed according to an associated mathematical probability density function or according to an associated discrete probability function
  - uniform int, uniform real, bernoulli, binomial, normal, …

```
std::default_random_engine e;           // a typedef for one of the engines
std::uniform_int_distribution<> d(1, 6);
d(e);                                   // returns an integer between 1 and 6
```

bind

A mechanism to transform an n-ary callable entity into an m-ary callable entity

```
char f(int i, double d);                // function
auto h = std::bind(f, 1, 2.);
h();                                    // calls f(1, 2.)
auto g = std::bind(f, std::placeholders::_1, 1.);
g(3);                                   // 3 is passed as _1 to f; calls f(3, 1.)
```

g has type `std::_Bind<char (*(std::_Placeholder<1>, double))(int, double)>` (in gcc)

```
struct T { char operator()(int); };  // function object
T t;
auto j = std::bind(t, 1);
j();                                    // calls t(1)

struct S { char foo(int k); };       // member function
S s;
auto k = std::bind(&S::foo, s, _1);
k(1);                                   // calls s.foo(1)
```

bind

- Useful for callbacks, operators/predicates/comparators to algorithms, …

```
default_random_engine e;
uniform_int_distribution<> d(1, 6);
d(e);                              // generates a random int
auto gen = bind(d, e);
gen();                             // calls d(e)

// generate 1000 numbers between 1 and 6 and store them into v
vector<int> v(1000);
generate_n(begin(v), v.size(), gen);
generate_n(begin(v), v.size(), bind(d, e));           // with bind
generate_n(begin(v), v.size(), [&] { return d(e); }); // with lambda

// check that all of them are less than 10
all_of(begin(v), end(v), [] (int i) { return i < 10; });  // with lambda
all_of(begin(v), end(v), bind(less<int>(), _1, 10));      // with bind
```

function

- A uniform wrapper that can hold any callable entity with a given signature

```
char foo(int);                                  // function
struct S { char foo(int); }; S s;               // member function
struct T { char operator()(int); }; T t; // function object

auto bf = bind(foo, _1);          // std::_Bind<char (*(std::_Placeholder<1>))(int)>
auto bs = bind(&S::foo, s, _1); // std::_Bind<std::_Mem_fn<char (S::*)(int) const> … >
auto bt = bind(t, _1);            // std::_Bind<T (std::_Placeholder<1>)>

std::function<char(int)> ff = bf;
std::function<char(int)> fs = bs;
std::function<char(int)> ft = bt;
ff(1); fs(1); ft(1);                            // callable like a function

char g(std::function<char(int)> f) { cout << f(1); }   // pass it to a function
std::map<int, std::function<char(int)>> int_to_f_map; // store in a data structure
```

Smart pointers: unique_ptr

- unique_ptr<T>, unique_ptr<T[]>
  - strict ownership
    - owns the object it points to and deletes it when it goes out of scope
  - movable, not copiable
  - replaces auto_ptr
  - supports a custom deleter

```
{
  unique_ptr<C> p(new C(…));
  C c = *p; // use like a pointer
  p->f();   // use like a pointer
  v.push_back(std::move(p));
  *p; // error
} // p automatically delete'd

{
  unique_ptr<C[]> q(new C[N]);
  q[i] = C();
} // q automatically delete []'d

{
  unique_ptr<char> p((char*)malloc(N), free);
} // the pointer will be deleted using free
```

Smart pointers: shared_ptr

- shared_ptr<T>
  - shared ownership
    - the last that goes out of scope deletes the object
  - movable and copiable
  - reference counted
  - supports a custom deleter
- if possible, use make_shared<> instead of direct new
  - faster (it saves an allocation)
  - exception safe
- no shared_ptr<T[]> yet

```
{
  shared_ptr<C> p(make_shared<C>(…)); // or
  auto p = make_shared<C>(…);
  C c = *p; // use like a pointer
  p->f();   // use like a pointer
  v.push_back(p);
  *p; // ok
} // the pointer is automatically deleted

{
  shared_ptr<FILE> f(
    fopen("main.cpp", "r"),
    fclose // custom deleter
  );
  fread(…, f.get());
} // fclose is automatically called
```

ratio

- Exact representation of any finite rational number with a numerator and denominator representable by compile-time constants
  - arithmetic operations
  - comparisons

```
typedef ratio<5, 3> x;
static_assert(x::num == 5 && x::den == 3, "x == 5/3");

typedef ratio<10, 6> y;
static_assert(y::num == 5 && y::den == 3, "y == 5/3");

static_assert(ratio_equal<x, y>::value == true, "5/3 == 10/6");

typedef ratio_divide<x, y>::type z;
static_assert(z::num == 1 && z::den == 1, "5/3 / 10/6 == 1");
```

ratio and SI

- Convenience typedefs

```
namespace std {
  typedef ratio<1, 1000000000000000000> atto;
  typedef ratio<1,    1000000000000000> femto;
  …
  typedef ratio<1,             1000000> micro;
  typedef ratio<1,                1000> milli;
  typedef ratio<1,                 100> centi;
  typedef ratio<1,                  10> deci;
  typedef ratio<                10, 1> deca;
  typedef ratio<               100, 1> hecto;
  typedef ratio<              1000, 1> kilo;
  typedef ratio<           1000000, 1> mega;
  …
  typedef ratio<  1000000000000000, 1> peta;
  typedef ratio<1000000000000000000, 1> exa;
}
```

chrono (time utilities)

- Duration

```
namespace std { namespace chrono {
template <class Representation, class Period = ratio<1>> class duration;
// convenience typedefs
typedef duration<int64_t, nano> nanoseconds;
typedef duration<int64_t>           seconds;
typedef duration<int, ratio<60>>    minutes;
…
}}

minutes m1(3);
minutes m2(2);
minutes m3 = m1 + m2;
microseconds us1(3);
microseconds us2(2);
microseconds us3 = us1 - us2;
microseconds us4 = m3 + us3;
minutes m4 = m3 + us3;       // error; to force use a duration_cast
us3 < us1;                   // true
```

chrono (time utilities)

- Clocks and time points
  - various clocks: system (wall-clock), steady (non decreasing), high resolution (with the shortest tick period)

```
chrono::system_clock::time_point start = chrono::system_clock::now();
this_thread::sleep_for(chrono::seconds(1));
cout << (chrono::system_clock::now() - start).count() << '\n';
// prints 1000131 (microseconds) on my machine with gcc

start = chrono::system_clock::now();
this_thread::sleep_until(start + chrono::seconds(1));
cout << (chrono::system_clock::now() - start).count() << '\n';
```

?