C++11

An Overview for C++98 Programmers

Dennis Klein

Scientific IT
GSI Darmstadt

Panda Computing Workshop 3-7 July 2017
Suranaree University of Technology (SUT), Nakhon Ratchasima

Outline

Part I

Part II

Part I

Wednesday, 5th July, 9.45am

Initialization

list-initialization

C++98

```
int a[4] = {1, 2, 3, 4};
int b = 1;
int c(2);
std::vector<int> d;
d.push_back(1);
d.push_back(2);
SomeObject e(1, 2);
```

list-initialization

C++98

```
int a[4] = {1, 2, 3, 4};
int b = 1;
int c(2);
std::vector<int> d;
d.push_back(1);
d.push_back(2);
SomeObject e(1, 2);
```

C++11

```
int a[4] = {1, 2, 3, 4};
int b = {1};
int c{2};
std::vector<int> d = {1, 2};

SomeObject e = {1, 2};
```

list-initialization

C++98

```
int a[4] = {1, 2, 3, 4};
int b = 1;
int c(2);
std::vector<int> d;
d.push_back(1);
d.push_back(2);
SomeObject e(1, 2);
```

C++11

```
int a[4] = {1, 2, 3, 4};
int b = {1};
int c{2};
std::vector<int> d = {1, 2};

SomeObject e = {1, 2};
```

new convention

Use braces for initialization everywhere.

list-initialization

C++98

```
int a[4] = {1, 2, 3, 4};
int b = 1;
int c(2);
std::vector<int> d;
d.push_back(1);
d.push_back(2);
SomeObject e(1, 2);
```

C++11

```
int a[4] = {1, 2, 3, 4};
int b = {1};
int c{2};
std::vector<int> d = {1, 2};

SomeObject e = {1, 2};
```

new convention

Use braces for initialization everywhere.

→But why?

avoids the vexing parse

```
TimeKeeper time_keeper(Timer());
```

avoids the vexing parse

```
TimeKeeper time_keeper(Timer());
```

Ambiguous:

1. A **variable declaration** with name `time_keeper` of type `TimeKeeper` that is initialized with an anonymous instance of `Timer`.
2. A **function declaration** with name `time_keeper` and return type `TimeKeeper` that accepts as arguments a single functor with return type `Timer` and no argument.

Compilers warn with `-Wvexing-parse` and interpret it as a function declaration.

avoids the vexing parse

```
TimeKeeper time_keeper(Timer());
```

Ambiguous:

1. A **variable declaration** with name `time_keeper` of type `TimeKeeper` that is initialized with an anonymous instance of `Timer`.
2. A **function declaration** with name `time_keeper` and return type `TimeKeeper` that accepts as arguments a single functor with return type `Timer` and no argument.

Compilers warn with `-Wvexing-parse` and interpret it as a function declaration. In C++11

1. `TimeKeeper time_keeper{Timer{}};`
2. `TimeKeeper time_keeper(Timer());`

more list-initialization

Aggregate initalization

```
struct Point {
    bool visited;
    double x, y, z;
}
Point p{false, 1., 2., 0.};
```

more list-initialization

Aggregate initalization

```
struct Point {
    bool visited;
    double x, y, z;
}
Point p{false, 1., 2., 0.};
```

Copy list-initialization with return

```
std::tuple<int, int> f() {
    // ...
    return {5, 25};
}
```

more list-initialization

Aggregate initalization

```
struct Point {
    bool visited;
    double x, y, z;
}
Point p{false, 1., 2., 0.};
```

Copy list-initialization with return

```
std::tuple<int, int> f() {
    // ...
    return {5, 25};
}
```

Member initializer list

```
struct A {
    A::A() : a{0}, b{42} {}

    private:
    int a, b;
}
```

more list-initialization

Aggregate initalization

```
struct Point {
    bool visited;
    double x, y, z;
}
Point p{false, 1., 2., 0.};
```

Copy list-initialization with return

```
std::tuple<int, int> f() {
    // ...
    return {5, 25};
}
```

Member initializer list

```
struct A {
    A::A() : a{0}, b{42} {}

    private:
    int a, b;
}
```

Brace elision

```
int m1[2][2] = {{1, 0}, {0, 1}};
int m2[2][2] = {1, 0, 0, 1};
```

std::initializer_list

```
#include <vector>
#include <initializer_list>

template <class T>
struct Wraptor {
    std::vector<T> v;
    Wraptor(std::initializer_list<T> l) : v(l) {}

    void append(std::initializer_list<T> l) {
        v.insert(v.end(), l.begin(), l.end());
    }
};

Wraptor<int> v1{1, 2, 3};     // direct list-initialization
Wraptor<int> v2 = {1, 2, 3}; // copy list-initialization
v2.append({4, 5});            // list-init in function call
```

constructor calls

```
struct Coord {
    Coord(int x, int y);
};
void foo(Coord, Coord);

Coord bar() {
    foo(Coord(1, 2), Coord(2, 3)); // C++98
    foo(Coord{1, 2}, Coord{2, 3}); // C++11
    foo({1, 2}, {2, 3});           // C++11
    return {3, 4};
}
```

auto

recap: templates

```
template<typename T> void foo(T) {}

foo(1);
foo(std::vector<float>{});
foo(foo<int>);
```

- The type `T` is deduced automatically by the compiler.
- The compiler creates three different functions `foo`.

auto type deduction for variables

```
auto x = 1;
auto y = std::vector<float>{};
auto z{42.};
```

equivalent to:

```
int x = 1;
std::vector<float> y = std::vector<float>{};
double z{42.};
```

auto type deduction for variables

```
auto x = 1;
auto y = std::vector<float>{};
auto z{42.};
```

equivalent to:

```
int x = 1;
std::vector<float> y = std::vector<float>{};
double z{42.};
```

There is much more to `auto` variables!
→ Recommended read: Chapter 1 Effective Modern C++ by Scott Meyers

trailing return type

```
auto pi() -> float { return 3.14159f; }
```

equivalent to:

```
float pi() { return 3.14159f; }
```

trailing return type

```
auto pi() -> float { return 3.14159f; }
```

equivalent to:

```
float pi() { return 3.14159f; }
```

But:

```
template<class T, class U>
auto add(T t, U u) -> decltype(t + u)
{ return t + u; }  // return type of operator+(T, U)
```

`auto` does not perform any type deduction here.

summary

Advice

Feel free to use `auto` often.

- Return types can be convoluted
- Implementation of generic functions becomes easier and more flexible

For loops

recap: for loops

```
for(int i = 0; i < 10; ++i) { ... }
```

→ perfectly fine

recap: for loops

```
for(int i = 0; i < 10; ++i) { ... }
```

→ perfectly fine

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

for(std::vector<int>::const_iterator i =
    v.begin(); i != v.end(); ++i) {
    int entry = *i;
    ...
}
```

→ cumbersome

recap: for loops

```
for(int i = 0; i < 10; ++i) { ... }
```

→ perfectly fine

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

for(std::vector<int>::const_iterator i =
    v.begin(); i != v.end(); ++i) {
    int entry = *i;
    ...
}
```

→ cumbersome, also `end()` is evaluated at every iteration!

recap: iterator loops

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

const std::vector<int>::const_iterator iEnd = v.end();
for(std::vector<int>::const_iterator i =
    v.begin(); i != iEnd; ++i) {
    int entry = *i;
    ...
}
```

→ *shudder*

range-based for loops

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

for(int entry : v) {
    ...
}
```

range-based for loops

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

for(int entry : v) {
    ...
}
```

General syntax: `for(rangeDeclaration : rangeExpression) statement`

rangeDeclaration type and variable name; may use auto; refs and const-refs are often useful

rangeExpression any expression that evaluates to a (temporary) object to be used as a sequence/range

statement same as with traditional for-loops.

range-based for loop internals

```
{
    auto&& __range = rangeExpression;
    for (auto __begin = begin_expr, __end = end_expr;
         __begin != __end; ++__begin) {
        rangeDeclaration = *__begin;
        statement
    }
}
```

__range	begin_expr	end_expr
array	`__range`	`__range + arraySize`
class w/ begin & end	`__range.begin()`	`__range.end()`
else	`begin(__range)`	`end(__range)`

more range-based for loops

```
int x[] = { 1, 2, 3 };
for (int a : x) {
    std::cout << a << std::endl;
}
```

more range-based for loops

```
int x[] = { 1, 2, 3 };
for (int a : x) {
    std::cout << a << std::endl;
}
```

```
for (auto a : { 3, 5, 7, 11 }) {
    std::cout << a << std::endl;
}
```

more range-based for loops

```
int x[] = { 1, 2, 3 };
for (int a : x) {
    std::cout << a << std::endl;
}
```

```
for (auto a : { 3, 5, 7, 11 }) {
    std::cout << a << std::endl;
}
```

```
std::vector<std::string> v{"one", "two", "three"};
for (const auto& s : v) {
    std::cout << s << std::endl;
}
```

Smart pointers

raw pointers are bad

Raw pointers were used/required for

- polymorphism
- shared access
- moving objects (without copy & delete)

with typical issues:

- resource leaks
- dereference of dangling pointers

ownership

Definition

A component *owns* an object, if it determines its lifetime.

ownership

Definition

A component *owns* an object, if it determines its lifetime.

`std::unique_ptr` Use for sole ownership

`std::shared_ptr` Use for shared ownership

`std::weak_ptr` Break cycles in shared ownership

std::unique_ptr

```
#include <memory>

class X {};

class Y : public X {};

void foo() {
    std::unique_ptr<Y> y{ new Y };
    std::unique_ptr<X> x = std::move(y); // transfer ownership
    // y is a nullptr now
    y = std::unique_ptr<Y>{static_cast<Y*>(x.release())};
    // x is a nullptr now
}   // Y::~Y() is called and the memory freed
```

requirement for safe usage

No assignment of raw pointers, unless very explicit:

```
Y* y = new Y;
unique_ptr<Y> a = y; // error
unique_ptr<Y> b; // points to nullptr
b = y; // error
b = {y}; // error
b = unique_ptr<Y>{y}; // ok, y still points
b = nullptr; // ok (Y::~Y() called and freed:
             //     y is dangling now!)
b = 0; // also works
```

std::shared_ptr

```
{
    std::shared_ptr<Y> y{new Y};
    // or:
    auto y = std::make_shared<Y>();

    auto x = std::static_pointer_cast<X>(y);
    // quit here and Y::~Y() called and freed once!
    std::shared_ptr<Y> z{y.get()}; // never do this
}   // Y::~Y() called and freed twice!
```

- does reference counting internally
- reference count can be queried via `use_count()`
- ref-counting works fine with casting

cycles

```
class B;

class A {
    std::shared_ptr<B> b;
};

class B {
    std::shared_ptr<A> a;
};
```

→ leak

- less obvious for larger cycles
- may be freed via call to `std::shared_ptr::reset()`
- better: `std::weak_ptr`

std::weak_ptr

```
class B;

class A {
    std::shared_ptr<B> b;
};
class B {
    std::weak_ptr<A> _a;
    void f() {
        if (auto a = _a.lock()) {
            // work with a
        } else {
            // the object was deleted
        }
    }
};
```

smart pointers to arrays

```
{
    std::unique_ptr<int[]> mem{ new int[100] };
    mem[0] = 0;
    mem[1] = 1;
    ...
} // calls delete[]
```

new idiom

new idiom

- never use owning raw pointers
- never use `delete` or `delete[]`

new idiom

new idiom

- never use owning raw pointers
- never use `delete` or `delete[]`

C++98:

```
Widget *widget = new Widget();
...
delete widget;
```

C++11:

```
auto widget = std::make_shared<Widget>();
\\ or
std::unique_ptr<Widget> widget{new Widget{}};
```

raw pointer are not so bad

```
class Node {
    std::vector<std::unique_ptr<Node>> children;
    Node* parent;
    ...
};
```

- `Node::parent` is by design not a dangling pointer
- the child does not own the parent

raw pointer are not so bad

```
class Node {
    std::vector<std::unique_ptr<Node>> children;
    Node* parent;
    ...
};
```

- `Node::parent` is by design not a dangling pointer
- the child does not own the parent

Remember

Note how the pointer type documents the ownership in C++11!

justify raw pointers

- document why a raw pointer is what you need
- or use a smart pointer

justify raw pointers

- document why a raw pointer is what you need
- or use a smart pointer

If you see `int* f()` complain!

- ownership semantics are undefined
- such an interface is asking for leaks or double frees
- this is a C interface, but not C++

Part II

Thursday, 6th July, 9.45am

Lambdas

functors

Definition

A *functor* is a record that stores a function.

functors

Definition

A *functor* is a record that stores a function.

```
#include <algorithm>

struct Between8And10 {
    template<typename T>
    bool operator()(const T& a) const {
        return a > T(8) && a < T(10);
    }
};

std::vector<double> v = { 8, 3.4, 5.1, 9, 43 };
if (std::any_of(begin(v), end(v), Between8And10())) {
    ...
}
```

closures

Definition

A *closure* is a record that stores a function and a stateful environment.

closures

Definition

A *closure* is a record that stores a function and a stateful environment.

```
struct Adder {
    int sum;
    adder() : sum(0) {}
    int operator()(int x) { return sum += x; }
}

Adder adder;
adder(3);
adder(4);
// adder.sum == 7
```

lambda (anonymous functor)

```
std::vector<double> v = { 8, 3.4, 5.1, 9, 43 };
if (std::any_of(begin(v), end(v),
    [](double x){ return x > 8. && x < 10.; }
)) {
    ...
}
```

lambda (anonymous functor)

```
std::vector<double> v = { 8, 3.4, 5.1, 9, 43 };
if (std::any_of(begin(v), end(v),
    [](double x){ return x > 8. && x < 10.; }
)) {
    ...
}
```

- define functors where they are used
- makes code easier to modify
- often easier to understand
- parts that belong together are placed together in the source

lambda captures (anonymous closure)

```
[&]{}; //ok: by-ref capture default
[=]{}; //ok: by-copy capture default
[&, i]{}; // ok: by-ref capt., except i is captured by copy
[=, &i]{}; // ok: by-copy capt., except i is captured by ref
[&, &i]{}; // error: by-ref capt. when by-ref is the default
[=, this]{}; // error: this when = is the default
[i, i]{}; // error: i repeated
[&, this]{}; // ok, but redundant

[&]() mutable { // non-const captures }
```

lambdas as arguments

```
template<typename F> void foo(F f) {
    f("Hello␣World.");
}

void foo(void (*f)(const char*)) {
    f("Hello␣World.");
}

void foo(std::function<void(const char*)> f) {
    f("Hello␣World.");
}
```

lambda return type

```
[ capture-list ]( args ) -> ret { body }
```

Return type ...

- *as specified*, or
- *implied* by the function return statements if not specified, or
- *void* if function body has no return statements

Multiple return statements must return the same type.

lambda type

The type of a lambda cannot be named, but can be inferred with auto

```
auto func1 = [](int i){ return i + 4; };
std::cout << "func1: " << func1(6) << std::endl;
```

Move semantics

the STL had a bad reputation

C++98:

```
struct X { ... };

void foo() {
    std::vector<X> v(5); // 1x X::X(), 5x X::X(const X&)
                         // 1x X::~X()
    std::vector<X> v2;
    v2 = v; // 5x X::X(const X&)
}
```

- `X::X()` called 1 times
- `X::X(const X&)` called 10 times
- `X::X()` called 1 times

the new STL was fixed

C++11:

```
struct X { ... };

void foo() {
    std::vector<X> v(5); // 5x X::X()
    std::vector<X> v2;
    v2 = std::move(v); // 0 calls to X
}
```

- `X::X()` called 5 times
- `X::X(const X&)` called 0 times
- `X::X()` called 0 times

lvalues/rvalues

- *lvalues* have a name
- *rvalues* are temporary objects
- you cannot take the address of an *rvalue*

```
int foo() { return 1; }

int bar() {
    //  lvalue = rvalue
    int x      = foo();
    int y      = x * x + 2;
    // -----
    return std::move(x);
    //     ^rvalue   ^lvalue
}
```

rvalue references

```
struct A { ... };

void foo(A&& a); // rvalue reference
void foo(A& a); // lvalue reference

void bar() {
    foo(A()); // calls foo(A &&)
    A a;
    foo(a); // calls foo(A &)
    foo(std::move(a)); // calls foo(A &&)
    // depending on the implementation of A,
    // further use of "a" may be undefined
}
```

forwarding references

```
template<typename T> void foo(T&& x);
```

- looks like an rvalue reference
- technically it is an rvalue reference
- it may behave as an lvalue reference
- used to be called universal reference (Scott Meyers)
- now it is called forwarding reference
- allows perfect forwarding

perfect forwarding

```
struct X {
    std::vector<Y> data;
    template<typename T> void add(T&& x) {
        // x is an lvalue!
        data.push_back(std::forward<T>(x));
        // calls either
        // * vector::push_back(const Y&) or
        // * vector::push_back(Y&&)
    }
};

void bar(X& x) {
    Y y;
    x.add(y); // adds a copy
    x.add(std::move(y)); // moves y into x.m_data
}
```

move semantics change design decisions

- it is now easier to design efficent value-based classes
- with C++98 one would sometimes use e.g. `std::vector<T>*` to avoid unnecessary (and expensive) copying

Take away

With move semantics proper C++11 code requires *fewer pointers*, and those should all be wrapped as *smart pointers*.

threadsafe singleton

- Singleton not recommended in new code, but still popular and widespread use
- Introduces global state, which limits optimizability, testability, maintainability, readability and scalability of your code.

threadsafe singleton

- Singleton not recommended in new code, but still popular and widespread use
- Introduces global state, which limits optimizability, testability, maintainability, readability and scalability of your code.

```
class X {
  public:
    static X& Instance() {
        static X instance;
        return instance;
    }

  private:
    X();
}
```

threadsafe singleton

- Singleton not recommended in new code, but still popular and widespread use
- Introduces global state, which limits optimizability, testability, maintainability, readability and scalability of your code.

```
class X {
  public:
    static X& Instance() {
        static X instance;
        return instance;
    }

  private:
    X();
}
```

threadsafe singleton

- Singleton not recommended in new code, but still popular and widespread use
- Introduces global state, which limits optimizability, testability, maintainability, readability and scalability of your code.

```
class X {
  public:
    static X& Instance() {
        static X instance;
        return instance;
    }

  private:
    X();
}
```

threadsafe singleton

- Singleton not recommended in new code, but still popular and widespread use
- Introduces global state, which limits optimizability, testability, maintainability, readability and scalability of your code.

```
class X {
  public:
    static X& Instance() {
        static X instance;
        return instance;
    }

  private:
    X();
}
```

STL concurrency

- `std::thread`
- `std::mutex`
- `std::lock_guard`
- `std::condition_variable`
- `future.get & promise.set_value`
- `std::atomic`
- Memory Model
- the new meaning of `const` and `mutable`: threadsafe

right angle bracket in templates

```
using std::vector;
vector<vector<int>> v; // error with C++98
```

- >> is one token in C++ and corresponds to the right shift operator
- in C++11 this is now valid code

long long

- the standard requires `sizeof(long) >= sizeof(int)`
- so we got: `long` is 32bit or 64bit depending on the target ABI
- C99 introduced `long long` to fix the issue:

```
sizeof(long long) >= sizeof(long) &&
sizeof(long long) >= 8
```

- with C++11 `long long` is now part of C++

default/delete functions

C++98:

```
class DontCopyMe {
    DontCopyMe(const DontCopyMe&);
    DontCopyMe& operator=(const DontCopyMe&);
  public:
    DontCopyMe() {}
    ...
};
```

C++11:

```
class DontCopyMe {
  public:
    DontCopyMe(const DontCopyMe&) = delete;
    DontCopyMe& operator=(const DontCopyMe&) = delete;
    DontCopyMe() = default;
    ...
};
```

virtual override/final

```
struct A {
    virtual void fill(int a) {}
};

struct B : public A {
    void fill(int a) override {}
    // or
    void fill(int a) final {}
};
```

If a virtual function is marked with the virt-specifier override and does not override a member function of a base class, the program is ill-formed.

nullptr

```
void foo(int*);
void foo(int);

void bar() {
    foo(0); // ambiguous
    foo(nullptr); // ok
}
```

new convention

Always use `nullptr`, never the literal `0` or the macro `NULL`.

user-defined literals

```
// used as conversion
constexpr long double operator"" _deg ( long double deg ) {
    return deg*3.141592/180;
}

int main(){
    double x = 90.0_deg;
    std::cout << std::fixed << x << std::endl;
}
```

- literal must begin with underscore
- literals ending with e or E need a space after when called

more

- `static_assert`
- `enum class`
- attributes
- type aliases, alias templates
- variadic templates
- `type_traits`
- `constexpr`
- ...

Acknowledgements

- Most slides in this presentation are based on a presentation by **Matthias Kretz**, GSI Darmstadt. His work is very much appreciated!

Material

- Beginner, Update for veterans: A Tour of C++ by Bjarne Stroustrup
- Advanced: Effective Modern C++ by Scott Meyers
- Because it came up in discussions: CppCon14: Modern Template Metaprogramming: A Compendium by Walter E. Brown