C++11: Selected Topics

Jörg Faschingbauer

www.faschingbauer.co.at

jf@faschingbauer.co.at

Table of Contents

FASCHINGBAUER

Overview

Make C++ Great Again

FASCHINGBAUER

C++ is one of the ugliest languages in the world

- Have to know C, including historical baggage
    - C preprocessor
    - No module concept
    - Implicit conversions
    - (*Many* more)
- No useful standard library
- Every new revision brings new features to solve old problems

C++11: The “New” C++

- Several years of development (since C++03)
- To be followed by C++14
- To be followed by C++17
- To be followed ...
- Focus
  - Easier usage (sometimes it reads like Python)
  - Performance

Overview

FASCHINGBAUER

Overview

FASCHINGBAUER

C++03 `enum` Types: Motivation

Why `enum`? Why isn't `int` sufficient?

- Readability, Semantics
- `switch` statements without `default` label → `-Wswitch` warns about missing enumerators
- Type safety: `int` cannot be assigned to an `enum`
  - The other way around is possible

Apart from that, `enum` is crap!

C++03 `enum` Types: Problems

- Enumerators are not in the `enum` type's scope
    - Rather, they pollute the surrounding scope
    - → no two enumerators with the same name
- Underlying type is not defined → `sizeof` depends on compiler
- Implicit conversion to `int`

Workarounds possible, although much typing involved!

C++11 `enum class`

`enum class`

```
enum class E1 {
  ONE,
  TWO
};
enum class E2 {
  ONE,
  TWO
};
E1 e1 = E1::ONE;
E2 e2 = E2::ONE;
int i = e1; // error
```

- No conflicts in surrounding scope
- Body same as before
- No conversion to `int`
- C++03 `enum` remains unchanged → code compatibility
- → Cool!

C++11 `enum class`: Underlying Type

`Explicite type`

```
#include <cstdint>
#include <cassert>
enum E: uint8_t {
  ONE,
  TWO
};
void f() {
  assert(sizeof(E)==1);
}
```

- In C++03 `enum` and `enum class` possible
- Default: `int`
- Works with every integer types except `wchar_t`

Overview

FASCHINGBAUER

auto Type Declarations: Motivation

FASCHINGBAUER

Much ado about nothing ...

```
vector<MyType>::iterator
  iter = v.begin();
```

Compiler knows anyway ...

```
auto iter = v.begin();
```

- *Type Deduction*
- Compiler knows anyway
- He always knew → lookup of template specializations
- → Same rules apply

`auto` Type Declarations: Details

Simplest Type Deduction

```
auto i = 10; // int
```

`cbegin()` → `const_iterator`

```
auto iter = v.cbegin();
```

`const` and References

```
const auto& cref = value;
```

Arrays are Pointers

```
int data[42];
// int *no_copy ...
auto no_copy = data;
```

Overview

Brace Initialization: Motivation (1)

Initialization was always inconsistent → Extremely confusing, especially for newbies!

- Integral types
- Aggregates (struct, array)
- Class objects
- Container (e.g. `std::vector`) initialization with contained values → `push_back()` orgies

Brace Initialization: Motivation (2)

FASCHINGBAUER

Integral Types

- Two different kinds of initialization
- A matter of history
- Initialization and assignment are different
- Constructor style necessary in templates → integers have to behave as if they were objects

Integer Initialization

```
int x = 7; // assignment style
int y(42); // ctor style
```

Brace Initialization: Motivation (3)

FASCHINGBAUER

Aggregates

- Initialization as it used to be in good old C
- No constructor style

Aggregate Initialization

```
int arr[] = {1, 2, 3};

struct s { int i,j; }
s s1 = {1, 2};
s s2 = {1}; // s2.j==0
```

Brace Initialization: Motivation (4)

Objects

- Constructor: looks like function call
- Copy initialization

Object Initialization

```
class C {
public:
  C(int i, int j);
};

C c1(1,2);
C c2 = c1;
```

Brace Initialization: Motivation (5)

FASCHINGBAUER

Containers

- Filling containers is extremely cumbersome → `.push_back()`
- *Initialization* requires an existing container → *very very loud*

Container Initialization

```
int arr[] = {1,2,3};
vector<int> v1(arr, arr+3);
vector<int> v2(v1.cbegin(), v1.cend());

set<int> s;
s.insert(1);
s.insert(2);
vector<int> v(s.cbegin(), s.cend());
```

Brace Initialization: Motivation (6)

Member Arrays

- *Cannot* be initialized
- Must be filled in constructor body
- → inconsistent
- → loud
- → workarounds

Member Array Initialization

```
class C {
public:
  C() : data_(/*dammit!*/) {}
private:
  const int data_[3];
};
```

Brace Initialization: Motivation (7)

Arrays on the Heap

- *Cannot* be initialized
- → inconsistent
- → loud
- → workarounds

Heap Array Initialization

```
const int *arr = new int[42];
// and now?
```

Brace Initialization: Solution (1)

Solution: brace initialization everywhere → the language becomes ...

- Clear
- Readable
- Memorizable (less exceptions)
- Attractive?

Brace Initialization: Solution (2)

Braces …

```
int i{42};

int arr[]{1,2,3};

struct s { int i,j; }
s s1{1,2};

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

… many more braces

```
class C {
public:
  C() : data_{1,2,3} {}
private:
  const int data_[3];
};

const int *arr =
  new const int[3]{1,2,3};
```

Overview

Range Based `for` Loops: Motivation (1)

`for` **looping over containers is very loud …**

- Iterators are cumbersome
- … albeit necessary
- `for_each` not always applicable
- → Why not building it into the language itself?

Range Based `for` Loops: Motivation (2)

Iteration, the cumbersome way

```
vector<int> v{1,2,3};
for (vector<int>::const_iterator it=v.begin();
     it!=v.end();
     ++it)
  cout << *it << endl;
```

This is cumbersome indeed ...

- `typedef` does not exactly help
- Iterators dereferenced by hand
- Much too loud

Range Based `for` Loops (1)

Solution: coupling the language with its standard library

The solution

```
vector<int> v{1,2,3};
for (int i: v)
  cout << i << endl;
```

Almost like Python, isn't it?

Range Based `for` Loops (2)

- Works with the usual `auto` incarnations …
- Valid for all C++ container types, arrays, initializer lists, etc.

`auto` Variants

```
vector<int> v{1,2,3};
for (auto& i: v) i = -i;
for (const auto& i: v)
  cout << i << endl;
```

Overview

FASCHINGBAUER

Delegating Constructor: Motivation

Every constructor does basically the same

```
class Data
{
public:
  Data(const void *p, size_t s) : data_(p), size_(s) {}
  Data(const string& s)
    : data_(s.c_str()), size_(s.size()) {}
private:
  const void *data_;
  size_t size_;
};
```

Delegating Constructor: Solution

Constructor *delegates*

```
class Data
{
public:
  Data(const void *p, size_t s) : data_(p), size_(s) {}
  Data(const string& s) : Data(s.c_str(), s.size()) {}
private:
  const void *data_;
  size_t size_;
};
```

Overview

“Return Object” Problem: Lifetime (1)

Whole class of problems: lifetime of returned objects

```
const std::string& f() {
  std::string s{"blah"};
  return s;
}
```

```
const std::string& f() {
  return "blah";
}
```

“Return Object” Problem: Lifetime (2)

```
const std::string& f() {
  std::string s{"blah"};
  return s;
}
```

- Object’s home is on the *stack*
- Returning *reference* to it
- → “undefined behavior”
- Fortunately compilers can detect and warn

```
warning: reference to local variable ‘s’ returned
     std::string s{"blah"};
                 ^
```

“Return Object” Problem: Lifetime (3)

```
const std::string& f() {
  return "blah";
}
```

- C string converted to `std::string` to match return type
    - Return type being *reference* is irrelevant
- → *temporary* object
- → “undefined behavior”

```
warning: returning reference to temporary
     return "blah";
            ^
```

"Return Object" Problem: Lifetime (4)

Solution: return by copy

```
std::string f() {
  return "blah";
}
```

- Before return, construct temporary from `"blah"`
- During return, copy-construct receiver object
- After return (during stack frame cleanup), destroy temporary
- → *Performance*
  - Though `std::string` objects are usually reference counted (but *not* by standard)
  - → Cheap copy

“Return Object” Problem: Performance

```
std::vector<int> f() {
  std::vector<int> v;
  int i=100000;
  while (i--)
    v.push_back(i);
  return v;
}
```

- Semantically correct
- Perfectly readable
- It’s just that arrays of 100000 elements aren’t copied so lightly
- Enter *Rvalue References*

(Teacher’s note: rvalueref-motivation.cc)

Move Semantics: Wish List

Wish list:

- Copy/assignment as before
- Special constructor for *moving*
- Can that be implemented in C++03?
    - Idea: non-const reference

Exercise

- Write a `class X` that carries an array of `int` and implements the usual copy semantics and a proper destructor.
- Additionally, for performance, the class provides a constructor that *transfers ownership* of the owned buffer.
- Try out the scenarios above, and see what's to be done in order for the *move constructor* to (not) be called.

Move Semantics, in C++03

Clumsy, isn’t it?

- Constructor with non-const reference preferred over const
- → Have to be explicit when moving is not wanted — *which is the regular case!*
-

Teacher’s notes:

- `moving-in-c++03.cc`
- In none of these use cases (except for function return) I want moving!
- Function return is optimized away → *Return Value Optimization (RVO)*

Lvalues and Rvalues (1)

```
int a = 42;
int b = 43;

a = b; // ok
b = a; // ok
a = a * b; // ok

int c = a * b; // ok
a * b = 42; // error, assignment to rvalue
```

Lvalues and Rvalues (2)

Rules ...

- Everything that has a name is an Lvalue
- Everything that I can assign to is an Lvalue
- Everything that I can take the address of is an Lvalue
- Everything else is an Rvalue

So ...

- Temporaries are clearly Rvalues
- As are function calls

Moving (1)

To make the long story short …

- Compiler will call `X(X&&)` when an Rvalue is passed
- E.g. function return
- Rules are far more complicated
- … as is the language
- (How about RVO?)

```
struct X
{
  X(X&& x)
  : data(x.data),
    size(x.size)
  {
    x.data = 0;
    x.size = 0;
  }

  int *data;
  size_t size;
};
```

Moving (2)

Compiler will DWIM ...

Return “by copy”

- Select `X(X&&)`
- Or RVO with copy ctor

```
X f()
{
  return X{"abc"};
}
X x = f();
```

Ordinary initialization

- Select `X(const X&)`

```
X x{"abc"};
X y = x;
```

Moving (3)

Explicitly requesting move operation

```
X y = std::move(x);
```

- `std::move` does not do anything the CPU must know
- Casts to `&&` to *force* selection of move-ctor
- Usage: `std::sort`, for example
    - Rearrange items
    - → Copy or move, depending on what’s there

No C++ Without Pitfalls

Compiler selects function based upon parameter type

- Normal overload selection
- Once called, the parameter *is an lvalue*
- Careful with moving

Bad

```
X(X&& x)
: s_(x.s_) {}
```

Good

```
X(X&& x)
: s_(std::move(x.s_))
```

Overview

`nullptr`

`NULL` is insufficient ...

- Typ is n ot defined
- Could be `void*`
- Or just as well `int`
- → Ambiguities

`nullptr`

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

f(NULL); // Hell!
f(nullptr); // f(int*)
```

Templates end with “>>”

Small parser insufficiency got fixed ...

> > vs. >>

```
std::map<int,vector<int> > ...;
std::map<int,vector<int>> ...; // C++11: THANK YOU!
```

→ **It’s about time!**

Overview

Why Smart Pointers?

Most prominent pointer (memory management) related bugs

- Memory leak
- Double free

Even more so with exceptions

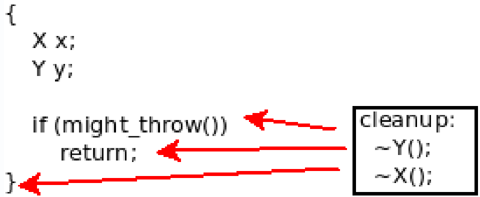
- Alternate return path
- Requires extra handling for resource cleanup

```
void do_something() {
    MyClass* tmp = new MyClass(666);
    do_something_with(tmp); // throws
    delete tmp;
    ...
}
```

Recap: Constructors and Destructors

Deterministic cleanup: at scope exit

- Explicit return
- End of scope
- Exceptions → *stack unwinding*



```
{
  X x;
  Y y;

  if (might_throw())
     return;
}
```

```
cleanup:
  ~Y();
  ~X();
```

Overview

FASCHINGBAUER

1. Introduction
2. New Language Features
   - Strongly Typed enum
   - auto Type Declarations
   - Brace Initialization
   - Range Based for Loops
   - Delegating Constructor
   - Moving, "RValue References"
   - Miscellaneous
3. Smart Pointers
   - std::unique_ptr<>
   - std::shared_ptr<>
   - Smart Pointers: Closing Words
4. Functions, Functions, ...
   - Optimization
   - Compute Bound Code
   - Basics
   - Sorting
   - std::bind
   - std::function
5. Multithreading
   - Threads Inroduction
   - Thread Life Cycle
   - Threads in C++
   - Race Conditions
   - Volatile
   - std::chrono
   - Locking and Atomics
   - Communication

Simplest: `std::unique_ptr<>`

```
#include <memory>
```

```
void do_something() {
    std::unique_ptr<MyClass> tmp(new MyClass(666));
    do_something_with(tmp.get());
    ...
}
```

- Destructor called at every return path
- No leaks

`std::unique_ptr<>`: Basic Usage

`std::unique_ptr<>` **is a pointer** → supports `->` and `*` operators in a natural way

```
ptr->do_something();
MyClass copy = *ptr;
```

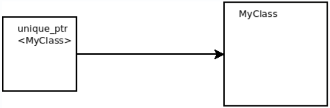
std::unique_ptr<>: Ownership (1)

Question: who is responsible to delete the object?

Answer:

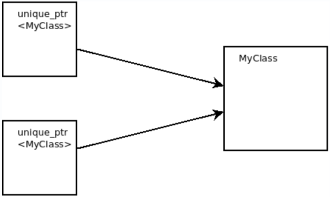
- If there is only one that points to it, then he's responsible
- If two point to it, then both are responsible

```
unique_ptr<MyClass> owner(
    new MyClass(666));
```



Just don't do it!

```
MyClass* tmp = new MyClass(666);
unique_ptr<MyClass> owner1(tmp);
unique_ptr<MyClass> owner2(tmp);
```



`std::unique_ptr<>`: Ownership (2)

Shared ownership: how else? → Copy!

```
unique_ptr<MyClass> owner(new MyClass(666));
unique_ptr<MyClass> another_owner = owner;
```

Compilation error

```
  ... error: use of deleted function ...
```

Good news ...

- `std::unique_ptr<>` is not copyable
- Only movable

`std::unique_ptr<>`: Ownership Transfer

"Unique" means that there can only be *one* owner

Passing a `std::unique_ptr<>`

```
void do_something_with(unique_ptr<MyClass> c);
void do_something()
{
    unique_ptr<MyClass> owner(new MyClass(666));
    do_something_with(owner);
}
```

Compilation error

```
error: use of deleted function  ... (copy) ...
```

std::unique_ptr<>: Ownership Transfer

FASCHINGBAUER

Back in C times ...

- Ownership conflict
- No solution but to be careful
- C++ 11: no implicit transfer when using smart pointers → compiler support for correctness
- → Clarity, no ambiguity

Explicit ownership transfer: `std::move`

```
void do_something_with(unique_ptr<MyClass> c);
void do_something()
{
    unique_ptr<MyClass> owner(new MyClass(666));
    do_something_with(std::move(owner));
    assert(owner == nullptr); // owner has given up ownership
}
```

std::unique_ptr<>: Juggling

Clearing

```
unique_ptr<MyClass> owner(new MyClass(666));
owner.reset(); // deletes object
```

Filling

```
unique_ptr<MyClass> owner;
owner.reset(new MyClass(666));
```

Stealing

```
unique_ptr<MyClass> owner(new MyClass(666));
MyClass* obj = owner.release();
```

`std::make_unique<>`: Pure Decadence

Lazyness

- C++ 11 brings lots of tools to save you keystrokes
- e.g. `auto`

`std::make_unique<>()`

```
auto ptr = make_unique<MyClass>(666);
```

Overview

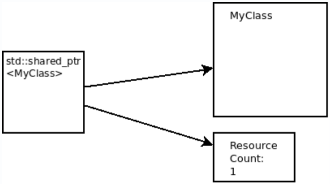
FASCHINGBAUER

`std::shared_ptr<>`: Not Unique

Ownership is not always clear ...

- Rare occasions where shared ownership is the right design choice
- ... laziness, mostly
- If in doubt, say `std::shared_ptr<>`

```
#include <memory>
std::shared_ptr<MyClass> ptr(
    new MyClass(666));
```

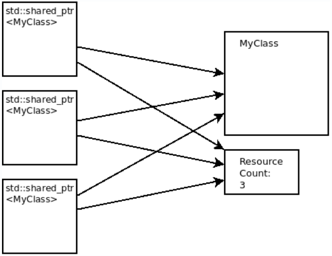


`std::shared_ptr<>`: Copy

FASCHINGBAUER

Copying is what shared pointer are there for

```
shared_ptr<MyClass> ptr(
    new MyClass(666));
shared_ptr<MyClass> copy1 = ptr;
shared_ptr<MyClass> copy2 = copy1;
```



`std::shared_ptr<>` vs. `std::unique_ptr<>`

How do `std::shared_ptr<>` and `std::unique_ptr<>` compare?

`std::unique_ptr<>`

- Small — size of a pointer
- Operations compile away entirely
- No excuse *not* to use it

`std::shared_ptr<>`

- Size of two pointers
- Copying manipulates the resource count
- Copying manipulates non-adjacent memory locations

`std::shared_ptr<>`: Object Lifetime

How long does the pointed-to object live?

- Reference count is used to track share ownership
- When reference count drops to zero, the object is *not referenced anymore*
- → deleted

Examining the reference count

```
shared_ptr<MyClass> ptr(new MyClass(666));
auto refcount = ptr->use_count();
```

Do not make any decisions based on it — at least not when the pointer is shared among multiple threads!

std::shared_ptr<>: Juggling

FASCHINGBAUER

Clearing: `reset()`

```
shared_ptr<MyClass> ptr(
    new MyClass(666));
auto copy = ptr;
ptr.reset();
```

- Decrements reference count
- Only if it becomes zero, object is deleted

Filling: `reset()`

```
shared_ptr<MyClass> ptr;
ptr.reset(new MyClass(666));
```

- Makes an empty pointer the initial reference

Overview

FASCHINGBAUER

Shared Pointers: Closing Words

Now when to use which pointer?

→ no definitive answer, but ...

Answer 1: performance, and designwise correctness

- Always use `std::unique_ptr<>` → clearly defined ownership
- Pass object around as pointer (`uptr->get()`)
- Use `std::shared_ptr<>` only if we have real shared ownership

Answer 2: programming efficiency

- Don't waste a thought on ownership, simply write it
- Always use `std::shared_ptr<>`

Overview

FASCHINGBAUER

Overview

FASCHINGBAUER

Optimization — Introduction

General Rules ...

- Focus on clean design → efficiency follows
- Optimization near the end of the project
- Proven hotspots need optimization
- *Proof through profiling*

“Premature optimization is the root of all evil”
Donald E. Knuth

Compute Bound or IO Bound? (1)

Decide whether, what and how to optimize!

- Collect representative input data
- Why does the program take long?
- Where does it spend most of its time?
  - Userspace: this is where computation is generally done
  - Kernel: ideally very little computation

Compute Bound or IO Bound? (2)

Checksumming From An Externel USB Disk

```
$ time sha1sum 8G-dev.img.xz > /dev/null
real 0m38.879s
user 0m3.349s
sys 0m0.375s
```

- `real`: total perceived run time (“wall clock time”)
- `user`: total CPU time spent in userspace
- `sys`: total CPU time spent in kernel

Here: `user` + `sys` is *far less* than `real` → mostly IO

Compute Bound or IO Bound? (3)

Checksumming From Internal SSD

```
$ time sha1sum 01\ -\ Dazed\ and\ Confused.mp3 1>/dev/null

real 0m0.128s
user 0m0.107s
sys 0m0.018s
```

Here: `user` + `sys` is *roughly equal* to `real`

- Almost no IO
- → Compute bound

What to do Next?

Now that we know that our application is compute bound …

- See where it spends most of its time → *profiling*
- Decide whether optimization would pay off
- Understand what can be done
- Understand optimizations that compilers generally perform

Overview

FASCHINGBAUER

Many Ways of Optimization

There are many ways to try to optimize code ...

- Unnecessary ones
- Using better algorithms (e.g. sorting and binary search)
- Function call elimination (inlining vs. spaghetti)
- Loop unrolling
- Strength reduction (e.g. using shift instead of mult/div)
- Tail call elimination
- ...

Unnecessary Optimizations

```
if (x != 0)
  x = 0;
```

- The rumour goes that this is not faster than unconditional writing
- Produces more instructions, at least

Inlining (1)

FASCHINGBAUER

Facts up front:

- Function calls are generally fast
- A little slower when definition is in a shared library
- Instruction cache, if used judiciously, makes repeated calls even faster
- But, as always: it depends

Possible inlining candidate

```
int add(int l, int r)
{
  return l + r;
}
```

Inlining (2)

A couple rules

- Always write clear code
- Never *not* define a function because of performance reason
  - *Readability first*
  - Can always inline later, during optimization
- Don't inline large functions → instruction cache pollution when called from different locations
- Use `static` for implementation specific functions → compiler has much more freedom

Inlining (3)

GCC ...

- Does not optimize by default
- Ignores explicit `inline` when not optimizing
- `-finline-small-functions` (enabled at `-O2`): inline when function call overhead is larger than body (even when not declared inline)
- `-finline-functions` (enabled at `-O3`): all functions considered for inlining → heuristics
- `-finline-functions-called-once` (enabled at `-O1`, `-O2`, `-O3`, `-Os`): all `static` functions that ...
- More → `info gcc`

Register Allocation (1)

- Register access is orders of magnitude faster than main memory access
  - → Best to keep variables in registers rather than memory
- CPUs have varying numbers of registers
  - `register` keyword should not be overused
  - Ignored anyway by most compilers
- Register allocation
  - Compiler performs flow analysis
  - Live vs. dead variables
  - “Spills” registers when allocation changes

Compiler generally makes better choices than the programmer!

Register Allocation (2)

GCC ...

- `-fira-*` (for Integrated Register Allocator)
- RTFM please
- A *lot* of tuning opportunities for those who care

Peephole Optimization

- **Peephole**: manageable set of instructions; “window”
- Common term for a group of optimizations that operate on a small scale
  - Common subexpression elimination
  - Strength reduction
  - Constant folding
- Small scale → “basic block”

Peephole Optimization: Common Subexpression Elimination

Sometimes one writes redundant code, in order to not compromise readability by introducing yet another variable ...

```
a = b + c + d;
x = b + c + y;
```

This can be transformed to

```
tmp = b + c; /* common subexpression */
a = tmp + d;
x = tmp + y;
```

Peephole Optimization: Strength Reduction

Most programmers prefer to say what they mean (fortunately) ...

```
x = y * 2;
```

The same effect, but cheaper, is brought about by ...

```
x = y << 1;
```

If one knows the “strength” of the operators involved (compilers tend to know), then even this transformation can be opportune ...

```
x = y * 3; /* y*(4-1) == y*4-y */
x = (y << 2) - y;
```

Peephole Optimization: Constant Folding

Another one that might look stupid but readable ...

```
x = 42;
y = x + 1;
```

... is likely to be transformed into ...

```
x = 42;
y = 43;
```

Consider transitive and repeated folding and propagation → pretty results

Loop Invariants

FASCHINGBAUER

The following bogus code …

```
while (1) {
  x = 42; /* loop invariant */
  y += 2;
}
```

… will likely end up as …

```
x = 42;
while (1)
  y += 2;
```

At least with a minimal amount of optimization enabled (GCC: `-fmove-loop-invariants`, enabled with `-O1` already)

Loop Unrolling

If a loop body is run a known number of times, the loop counter can be omitted.

```
for (i=0; i<4; i++)
  dst[i] = src[i];
```

This can be written as

```
  dst[0] = src[0];
  dst[1] = src[1];
  dst[2] = src[2];
  dst[3] = src[3];
```

- *Complicated heuristics*: does the performance gain outweigh instruction cache thrashing?
- → I'd keep my fingers from it!

Tail Call Optimization

```
int f(int i)
{
   do_something(i);
   return g(i+1);
}
```

- `g()` is called *at the end*
- `f()`'s stack frame is not used afterwards
- **Optimization:** `g()` can use `f()`'s stack frame

CPU Optimization, Last Words

Once more: **Write clean Code!**

- All optimization techniques explained are performed *automatically*, by the compiler
- Theory behind optimization is well understood → engineering discipline
- Compilers generally perform optimizations better than a programmer would
  - ... let alone portably, on different CPUs!
- “Optimization” is a misnomer → “Improvement”
  - Compiler cannot make arbitrary code “optimal”
  - Bigger picture is always up to the programmer
  - → Once more: **Write clean Code!**
- Work together with compiler → use `static`, `const`

GCC: Optimization “Levels”

- `-O0`: optimization off; the default
- `-O1`: most basic optimizations; does as much as possible without compromising compilation time too much
- `-O2`: recommended; does everything which has no size impact, is unagressive, and doesn’t completely chew compilation time
- `-O3`: highest level possible; somewhat agressive, can break things sometimes, eats up your CPU and memory while compiling
- `-Os`: optimize for size; all of `-O2` that doesn’t increase size
- `-Og` (since GCC 4.8): “developer mode”; turns on options that don’t interfere with debugging or compilation time

Overview

FASCHINGBAUER

Containers, Iterators, Algorithms

Genius Combination of ...

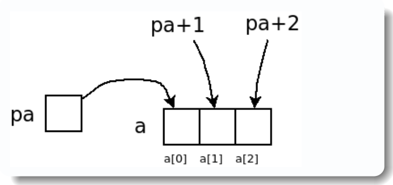
- Operator overloading (->, $*$, +, +=, ++)
- Abstract containers
- Abstract “Algorithms”
- ... based upon *pointer arithmetic*!

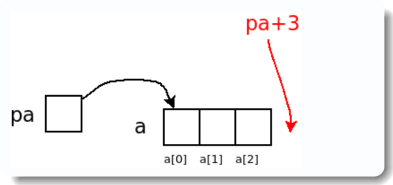
→ *Pointer arithmetic*, revisited ...

Pointer Arithmetic (1)

Pointer and arrary index

- *Pointer + Integer = Pointer*
- Exactly the same as subscript (“index”) operator
- *No range check*
- → Error prone
- But: performance!

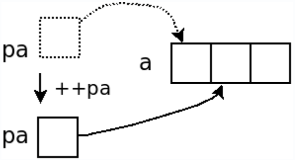




Pointer Arithmetic (2)

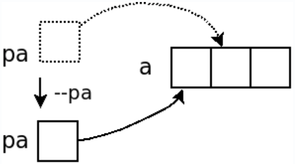
Pointer Increment

```
int *pa = a;
++pa;
```



Pointer Decrement

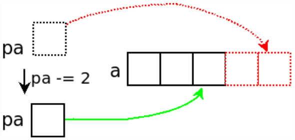
```
int *pa = &a[1];
--pa;
```



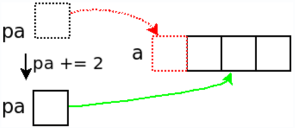
Pointer Arithmetic (3)

Pointer don't necessarily point to valid memory locations …

```
*pa = a + 4;
pa -= 2;
i = *pa; /* ok */
```



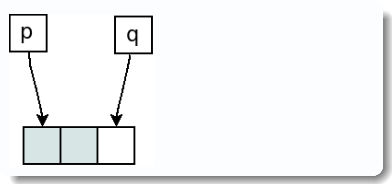
```
*pa = a - 1;
pa += 2;
i = *pa; /* ok */
```



Pointer Arithmetic: Difference

How many array elements are there between two pointers?

```
p = &a[0];
q = &a[2];
num = q - p; /* 2 */
```



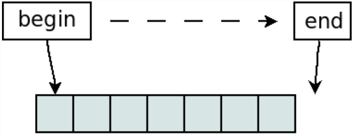
General practice ("The Spirit of C"):

- *Beginning* of an array (a *set* of elements) is a *pointer to the first element*
- *End* is *pointer past the last element*

Pointer Arithmetic: Array Algorithms

Iteration over all elements of an array ...

```
int sum(const int *begin, const int *end)
{
    int sum = 0;

    while (begin < end)
        sum += *begin++; /* precedence? what? */
    return sum;
}
```



Pretty, isn't it?

Pointer Arithmetic: Step Width? (1)

So far: pointer to int `int` — how about different datatypes?
→ same!

- *pointer* + *n*: points to the *n*-th array element from *pointer*
- Type system knows about sizes
- Pointer knows the type of the data it points to
- Careful with `void` and `void*`

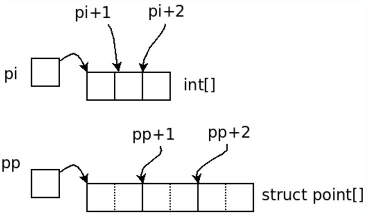
Pointer Arithmetic: Step Width? (2)

```
struct point
{
    int x, y;
};

struct point points[3], *begin, *end;

begin = points;
end = points + sizeof(points)/sizeof(struct point);

while (begin < end) {
    ...
    ++begin;
}
```

Pointer Arithmetic: Arbitrary Data Types?

- *sizeof*: size (in bytes) of a type or variable

```
sizeof(int)
sizeof(struct point)
sizeof(i)
sizeof(pi)
sizeof(pp)
```



Container

Container

- Extremely practical collection of template classes
- Sequential container → array, list
- Associative containers

Dynamically growing array: `std::vector`

```
#include <vector>

std::vector<int> int_array;
int_array.push_back(42);
int_array.push_back(7);
int_array.push_back(666);

for (int i=0; i<int_array.size(); ++i)
    std::cout << int_array[i] << ' ';
```

Pointer Arithmetic

```
std::vector<int>::const_iterator begin = int_array.begin();
std::vector<int>::const_iterator end = int_array.end();
while (begin < end) {
    std::cout << *begin << ' ';
    ++begin;
}
```

Algorithms: `std::copy` (1)

Copy array by hand

```
std::vector<int> int_array;
int_array.push_back(42);
int_array.push_back(7);
int_array.push_back(666);

int int_array_c[3];
std::vector<int>::const_iterator src_begin = int_array.begin()
std::vector<int>::const_iterator src_end = int_array.end();
int *dst_begin = int_array_c;

while (src_begin < src_end)
    *dst_begin++ = *src_begin++;
```

Algorithms: `std::copy` (2)

FASCHINGBAUER

Copy using STL

```
#include <algorithm>

std::vector<int> int_array;
// ...
int int_array_c[3];

std::copy(int_array.begin(), int_array.end(), int_array_c);
```

Adapting Iterators: `std::ostream_iterator`

FASCHINGBAUER

Copy: array to `std::ostream`, which looks like another array

```
#include <iterator>

int int_array_c[] = { 34, 45, 1, 3, 2, 666 };
std::copy(int_array_c, int_array_c+6,
          std::ostream_iterator<int>(std::cout, " "));

std::vector<int> int_array;
// ...
std::copy(int_array.begin(), int_array.end(),
          std::ostream_iterator<int>(std::cout, " "));
```

Adapting Iterators: `std::back_insert_iterato`

Problem

- `std::copy()` requires *existing/allocated memory* → *performance!*
- → copying onto empty `std::vector` impossible

Segmentation Fault

```
int int_array_c[] = { 34, 45, 1, 3, 2, 666 };
std::vector<int> int_array; // empty!

std::copy(int_array_c, int_array_c+6, int_array.begin());
```

Adapting Iterators: `std::back_insert_iterato`

Solution: `std::back_insert_iterator`

```
int int_array_c[] = { 34, 45, 1, 3, 2, 666 };
std::vector<int> int_array;

std::copy(
    int_array_c, int_array_c+6,
    std::back_insert_iterator<std::vector<int> >(int_array));
```

Overview

Algorithms: `std::sort`

FASCHINGBAUER

Now for something simple ...

C

```
int int_array[] = { 34, 45, 1, 3, 2, 666 };
std::sort(int_array, int_array + 6);
```

C++

```
std::vector<int> int_array;
int_array.push_back(42);
int_array.push_back(7);
int_array.push_back(666);

std::sort(int_array.begin(), int_array.end());
```

Algorithms: `std::sort`, custom comparison

```
bool less_reverse(int l, int r)
{
    return l > r;
}

int int_array[] = { 34, 45, 1, 3, 2, 666 };
std::sort(int_array, int_array + 6, less_reverse);
```

Overview

`std::bind`: Why?

Why? What's the problem?

Answer:

- Hard to explain
- Best to see the problem first
- Let's start small, by simple example

Problem: we have ...

- Two dimensional points `(x,y)`
- A function to compute the distance between two points

We want:

- A function to compute the distance from *origin* `(0,0)`

What We Have

Point

```
struct Point
{
  Point(double x, double y)
    : x(x), y(y) {}
  double x, y;
};
```

Distance

```
double distance(Point p, Point q)
{
  return std::sqrt(
    std::pow(std::abs(p.x-q.x), 2) +
    std::pow(std::abs(p.y-q.y), 2)
  );
}
```

Retro C/C++

- We have all that is needed
- Could easily define a small function
- → Problem solved
- *But this would be soo retro!*

Distance from Origin

```
double distance_origin(Point p)
{
  return distance(p, {0,0});
}
```

The Real Problem

Nothing is wrong with small functions

- Compiler will inline them
    - … and optimize away entirely
- Defined centrally (public header file?) for further reuse

But…

- What if they serve *only one* purpose?

Sample Problem

Compute the origin-distances of an array of points, and store those in an equally sized array of `double`!

Straightforward Implementation

Near the top of the implementation file …

One-Time Function Definition

```
static double distance_origin(Point p) {
    return distance(p, {0,0});
}
```

And *far down below*, in the implementation section …

Location of use

```
double distances_origin[sizeof(swarm)/sizeof(Point)];
std::transform(swarm, swarm+sizeof(swarm)/sizeof(Point),
               distances_origin,
               distance_origin);
```

More Sample Problems

Another Sample Problem

Compute the distances of an array of points from a given point, and store those in an equally sized array of `double`!

Possible solutions: as many as there are different tastes around ...

- Lets write another stupid function, basically a copy of `distance_origin` — only with `(1,1)` instead of `(0,0)`
- Even better: lets generalize! *Functors! Function call operator!*

More Straightforward Implementations

One-Time Functor Definition

```
struct distance_point {
  distance_point(Point origin) : origin(origin) {}
  double operator()(Point p) const {
      return distance(p, origin);
  }
  Point origin;
};
```

Location of use

```
double distances_origin[sizeof(swarm)/sizeof(Point)];
std::transform(swarm, swarm+sizeof(swarm)/sizeof(Point),
               distances_point,
               distance_point({1,1}));
```

Readability

Provided that the helper code is only used once ...

- *Readability* is inversely proportional to amount of code
- *Number of bugs* is directly proportional to amount of code
- Helper implementation is nowhere near location of use
- `static` is the only keyword that enhances readability

Similar problem with many data structures and algorithms ...

- Sorting criteria: `std::sort`, `std::map`, ...
- Predicates: `std::find_if`, `std::equal`, ...
- Arbitrary adaptations where helper functions are needed
  - Most prominent (although relatively useless nowadays): `std::for_each`

Introducing std::bind (1)

Best done by example …

```
void f(int a, int b) {
  std::cout << a << ',' << b << std::endl;
}
```

Direct function call

```
f(1, 2);
```

prints …

```
1,2
```

What if we need the functionality of `f(a, b)`, but are required to pass a *callable* that taken no parameters?

Introducing `std::bind` (2)

In other words, we need to create a function-like object that wraps `f(a,b)` that always calls `f` with, say, `a=1` and `b=2`.

Hardcoded parameters

```
auto bound = std::bind(f, 1, 2);
bound();
```

prints …

```
1,2
```

- Alternative: manually write function adaptor (functor) that remembers parameters until called
- Origin: Boost (www.boost.org)

Introducing `std::bind` (3)

FASCHINGBAUER

Routing parameters into arbitrary positions: `std::placeholders`

Hardcoding only second parameter

```
auto bound = std::bind(f,
        42, std::placeholders::_1);
bound(7);
```

prints ...

```
42,7
```

Exchanging parameters

```
auto bound = std::bind(f,
        std::placeholders::_2,
        std::placeholders::_1);
bound(1,2);
```

prints ...

```
2,1
```

Applying `std::bind` (1)

So how does this apply to our `std::transform` problem?

- Readability: we want to eliminate those annoying extra helper functions
- Want to wrap existing `double distance(Point, Point)` which is similar in purpose but does not fit exactly

What we have ...

```
struct Point {...};
double distance(Point, Point);
```

What we want ...

```
std::transform(swarm, swarm+sizeof(swarm)/sizeof(Point),
               distances_point,
               SOMETHING WHICH TAKES ONE POINT);
```

Applying `std::bind` (2)

FASCHINGBAUER

Distances from origin

```
std::transform(swarm, swarm+sizeof(swarm)/sizeof(Point),
               distances_origin,
               std::bind(distance,
                  Point{0,0}, std::placeholders::_1));
```

Distances from any point

```
// this is exactly the same as above
```

Summary

- Readability: what remains unreadable is only the language itself
- Have to get used to `std::bind`

std::bind vs. Lambda

FASCHINGBAUER

Lambdas are usually a better alternative …

```
std::transform(swarm, swarm+sizeof(swarm)/sizeof(Point),
               distances_origin,
               [](Point p) { return distance({0,0}, p); });
```

A more advanced exercise

Use `std::sort` to sort an array of points by their distance to a given point.

A Bigger Picture: Types

FASCHINGBAUER

What about types?

- Goal is to have *no runtime overhead*
- $\Longrightarrow$ *Late binding (polymorphism)* ruled out
- $\Longrightarrow$ No common base class
- Only the call signatures (parameter and return types) are the same

What does this mean?

- Perfect for `<algorithm>` which is also designed for speed
- Have to be careful when code size is important
- Client code has to be instantiated with the type
- **Tradeoff**: speed, code size, elegance, design, taste ...

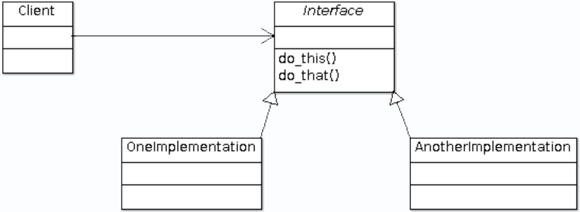
Overview

Classic Polymorphism

Back to classic Object Oriented Design …

- **Interfaces** *define* what methods have to be available on an object
- **Implementations** *provide* those methods
- **Clients** *use* **interfaces**

(Teacher's note: `classic-polymorphism.cc`)



Classic Polymorphism: Upsides

Polymorphism is well understood:

- *Late binding*: client does not know the exact type that is being used
- *Interfaces* describe relationships in almost human language — *if done right*
- *Software Architecture* — *if done right* — is almost self-explanatory
- *Design Patterns* are described (and mostly implemented as well) in such a way
- Also available in other languages
  - For example Java explicitly distinguishes between *interface* and *implementation*

Classic Polymorphism: Technical Downsides

There are purely technical downsides (in C++ at least)

- Runtime overhead
  - Not knowing the exact type implies *indirect call* (function pointer/trampoline)
- Code size
  - If one writes `virtual`, a whole bunch of code is generated (Runtime Type Information — RTTI)
  - Type is not POD (*plain old data*) anymore

Classic Polymorphism: More Downsides

Metaphysical downsides are harder to come by: **readability** again

- Provided that logging has no architectural relevance ...
- I have two functions which are similar in purpose, but otherwise unrelated. How can I arrange for client code to use these interchangeably?
    - Why can't I *just use* them?
    - I don't want to instantiate client code from a template!
    - Do I really want to craft an interface for client code to use?
- I have a class that has similar purpose as the functions
    - Client code wants to just call it
- I want to *adapt* all these!
- Sound like the solution is `std::bind`
- → Wrong: `std::bind` objects don't share a type

(Teacher's note: `classic-polymorphism-logger.cc`)

`std::function` to the Rescue (1)

- **One type** to rule them all!
- → *Any* callable with same signature

Function object

```
std::function<int(int, int)> foo_func;
```

Trivial: plain function

```
int foo(int a, int b) { ... }
foo_func = foo;
```

`std::function` to the Rescue (2)

Any `std::bind` object

```
struct bar {
    int foo(int a, int b) { ... }
};
foo_func = std::bind(&bar::foo, &bar,
        std::placeholders::_1, std::placeholders::_2);
```

Lambda

```
foo_func = [](int a, int b) -> int { ... };
```

`std::function`: Last Words

Upsides

- *Lightweight Polymorphism*: no code explosion
- Unlike *heavyweight polymorphism*, no dynamic allocation appropriate
    - Although a `std::function` object can hold polymorphic callables, it is always the same size

Downsides

- *Runtime overhead* due to indirect call
    - Processor support makes them just as fast as direct function calls
    - *But:* no inlining possible
- *Readability* again ...
    - This is not OO!
    - *Architectural intentions* not at all obvious through quick inline adaptations

Overview

FASCHINGBAUER

1. Introduction
2. New Language Features
   - Strongly Typed `enum`
   - `auto` Type Declarations
   - Brace Initialization
   - Range Based `for` Loops
   - Delegating Constructor
   - Moving, “RValue References”
   - Miscellaneous
3. Smart Pointers
   - `std::unique_ptr<>`
   - `std::shared_ptr<>`
   - Smart Pointers: Closing Words
4. Functions, Functions, ...
   - Optimization
   - Compute Bound Code
   - Basics
   - Sorting
   - `std::bind`
   - `std::function`
5. Multithreading
   - Threads Inroduction
   - Thread Life Cycle
   - Threads in C++
   - Race Conditions
   - Volatile
   - `std::chrono`
   - Locking and Atomics
   - Communication

Overview

FASCHINGBAUER

Operating System Primitives

- C++ does not *implement* threads
- They only wrap OS primitives
    - POSIX Threads → `man pthreads`
    - Windows → MSDN
    - Embedded OSen?

There Be Dragons

Threads are the work of the devil!

- Everything that used to be correct in a singlethreaded world is questionable in the face of threads
- *Race conditions*, even in the most innocent looking places

Corollary:

- A project that was designed without threads in mind is useless with threads
- Multithreading has to be planned *from the beginning*
- Creation of a new thread must be justified to God

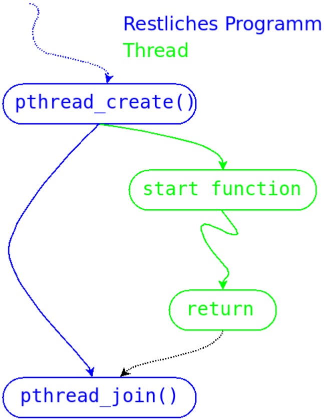
That being said ...

Overview

Thread Life Cycle

- `pthread_create()` creates new thread
- *Start function* is called
- Thread terminates
- `pthread_join()` synchronizes with termination (fetches “exit status”)

No parent/child relationship → anybody can join



Thread Creation

FASCHINGBAUER

man 3 pthread_create

```
int pthread_create(
    pthread_t *thread, const pthread_attr_t *attr,
    void *(*start_routine) (void *), void *arg);
```

- `thread`: ID of the new thread (“output” parameter)
- `attr` → see later (`NULL` → default attribute)
- `start_routine`: thread start function, `void*/void*`
- `arg`: parameter of the start function

Thread Termination (1)

Thread termination alternatives:

- Return from start function
- `pthread_exit()` from somewhere inside the thread (cf. `exit()` from a process)
- `pthread_cancel()` from outside (cf. `kill()`)
- `exit()` of the entire process → all contained threads are terminated

Don't use `pthread_cancel()` unless you know what you are doing!

Thread Termination (2)

Without any further ado: the manual …

man 3 pthread_exit

```
void pthread_exit(void *retval);
```

man 3 pthread_cancel

```
int pthread_cancel(pthread_t thread);
```

Exit Status, `pthread_join()`

A thread's "exit status":

- `void*`, just like the start parameter → more flexible than a process's `int`.
- Parameter to `pthread_exit()`
- Return type of the start function

man 3 pthread_join

```
int pthread_join(pthread_t thread, void **retval);
```

Detached Threads

Sometimes one does not want to use `pthread_join()`

- Rather, run a thread in the “background”.
- “Detached” thread
- Thread attribute

man 3 pthread_attr_setdetachstate

```
int pthread_attr_setdetachstate(
    pthread_attr_t *attr, int detachstate);
PTHREAD_CREATE_DETACHED
  Threads that are created using attr will be created in a
  detached state.
```

- Detaching at runtime ...

man 3 pthread_detach

```
int pthread_detach(pthread_t thread);
```

Thread ID

- `pthread_create()` returns `pthread_t` to the caller
- Thread ID of calling thread: `pthread_self()`
- Compare using `pthread_equal()`

man 3 pthread_self

```
pthread_t pthread_self(void);
```

man 3 pthread_equal

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

“Scheduled Entities” (1)

Kernel maintains “scheduled entities” (Process IDs, “1:1” scheduling)

Threads inside `firefox`

```
$ ps -eLf|grep firefox
$ ls -1 /proc/30650/task/
13960
13961
... (many more) ...
```

“Scheduled Entities” (2)

Too bad:

- Scheduled entity’s ID *is not the same as* `pthread_t`
- Correlation of OS threads and POSIX thread is Linux specific

man 2 gettid

```
pid_t gettid(void);
```

Overview

FASCHINGBAUER

Creating Threads is Far Too Easy

FASCHINGBAUER

No parameterization

```
void f() { ... }
std::thread t(f);
```

`std::bind`?

```
void f(int i) { ... }
std::thread t(f, 666);
```

Lambdas

```
std::thread t([](){ ... });
```

Looks all pretty familiar, no?

Joinable vs. Detached

Why wait for termination?

- Wait for a calculation to finish
    - Distribute parallelizable algorithm over multiple CPUs
- Graceful program termination

Synchronize caller with termination of `t`

```
t.join();
```

Why detach a thread?

- Background service thread → program lifetime

Detach a thread

```
t.detach();
```

Cornercases in Thread Lifetime

What if the program terminates before a thread?

```
int main() { std::thread t([](){for(;;);}); }
```

On Linux, at least ...

- When a process terminates, all its threads terminate *immediately*

Can I terminate a thread without its cooperation?

- In Linux, yes, theoretically
- What happens with locked mutexes?
- → Cancellation hooks (hell!)

Portably, no!

Overview

Exercises: Thread Creation, Race Condition

- Write a program that creates two threads. Each one of the threads increments *the same* integer, say, 10000000 times.
  - The integer is shared between both threads (allocated in the `main()` function). A pointer to it gets passed to the thread start function.
  - The threads don't increment a copy of the integer, but rather access *the same* memory location.

After the starting process (the *main thread*) has synchronized with the incrementer's termination, he outputs the current value of the said integer.
What do you notice?

Race Conditions (1)

Suppose `inc()` is executed by at least two threads in parallel:

Very bad code

```
static int global;

void inc()
{
    global++;
}
```

CPU A		CPU B		
Instr	Reg	Instr	Reg	Mem
load	42	load	42	42
inc	43	inc	43	42
	43	store	43	43
store	43		43	43

- *The variable `global` has seen only one increment!!*
- “Load/Modify/Store Conflict”
- The most basic race condition

Race Conditions (2)

Imagine more complex data structures (linked lists, trees): if incrementing a dumb integer bears a race condition, then what can we expect in a multithreaded world?

- No single data structure of C++'s Standard Template Library is thread safe
- `std::string`'s copy construktor and assignment operator are thread safe (*GCC's Standard C++ Library* → *not* by standard)
- `std::string`'s other methods are *not* thread safe
- *stdio* and *iostream* are thread safe (by standard since C++11)

Overview

`volatile`: The Lie (1)

What `volatile` does:

- Prevents *compiler* optimization of everything involving the variable declared `volatile`
- Corollary: the variable must not be kept in a register

```
volatile int x;
```

Attention:

- All it does is provide a false impression of correctness
- Most of its uses are outright bugs

`volatile`: The Lie (2)

What `volatile` **doesn't:**

- Variable can still be in a cache
    - *Variable is not at all sync with memory* when using *write-back* cache strategy
- Not a memory barrier → load/store reordering still possible (done by CPU, *not by compiler*)
- → *Not a replacement for proper locking*

Still broken: *load-modify-store*

```
volatile int use_count;

void use_resource(void)
{
  do_something_with_shared_resource();
  use_count++;
}
```

`volatile`: Valid Use: Hardware

Originally conceived for use with hardware registers

- Optimizing compiler would wreak havoc
    - Loops would never terminate
    - Memory locations would not be written to/read from
    - ...

```
volatile int completion_flag;
volatile int out_word;
volatile int in_word;

int communicate(int word)
{
    out_word = word;
    while (!completion_flag);
    return in_word;
}
```

`volatile`: Valid Use: Unix Signal Handlers

A variable might change in unforeseeable ways

- Signal handler modifies `quit` variable
- Optimizing compiler would otherwise make the loop endless

```
volatile int quit;

int main(void)
{
  while (!quit)
    do_something();
}
```

Overview

FASCHINGBAUER

`std::chrono`

Time is complex

- … and so is `std::chrono`
- Time points, starting at Epoch
    - E.g. Good (?) old `time_t`, in seconds since 1970-01-01 00:00:00
- Multiple *clock domains*, each with their own notion of time points (varying in epoch and time unit)
- Duration
    - Difference between time points
    - Time point — duration between time point's epoch and itself

Clock Domains

- `system_clock`
  - "Wall clock time", based upon the system's notation of time.
  - Unix: `time_t`, starting 1970-01-01, in seconds.
  - Not monotonic — modified by e.g. NTP
- `steady_clock`
  - Starts at arbitrary timepoint — commonly system boot
  - *Monotonic*: advances steadily
  - E.g. POSIX's `CLOCK_MONOTONIC`
- `high_resolution_clock`
  - "High resolution timers" — ultimately, this is "brand new hardware"
  - Usually used to formulate high-precision wait periods etc.

Measuring Time (1)

A snapshot of time: a clock domain's `time_point`

Now

```
#include <chrono>

std::chrono::system_clock::time_point now =
     std::chrono::system_clock::now();
```

Measuring Time (2)

Duration: difference between points

Duration

```
std::chrono::steady_clock::duration spent = after - before;
std::chrono::milliseconds spent_milli =
    std::chrono::duration_cast<std::chrono::milliseconds>
         (spent);
std::cout << spent_milli.count() << std::endl;
```

Note: use `steady_clock` time points to compute intervals — other clock are not immune against time modifications

Overview

FASCHINGBAUER

Mutex

Exclusive lock

- Can be taken by *only one thread*
- Methods:
    - lock: take (and possibly wait for) lock
    - unlock
    - try lock: take lock, or return error if locked

```
#include <mutex>
std::mutex lock;

lock.lock();
... critical section ...
lock.unlock();
```

Scoped Locking (1)

What if a critical section throws?

```
lock.lock();
do_something_errorprone(); // possibly throws
do_more_of_it(); // possibly throws
lock.unlock();
```

- Lock remains locked
- → Deadlock

Scoped Locking (2)

Deterministic destructors

- Objects are destroyed *at end of block*
- Unlike Java, Python, ... (garbage collection)
- → **Exception safety!**

std::lock_guard

```
...
// critical section
{
  std::lock_guard<std::mutex> g(lock); // lock.lock()
  do_something_errorprone();
  do_more_of_it();
  // ~guard does lock.unlock();
}
...
```

Mutex: Pros and Cons

Mutexes are heavyweight

- *Context switch* on wait → expensive
- Can only be used in thread context
- Interrupts *cannot wait*
- → *Never share mutexed objects with an interrupt routine!*
- → *Undefined behavior*

Mutexes are easy

- Can protect arbitrarily long critical sections

Atomic Instructions (1)

Simple integers don't need a mutex → *atomic instructions*

GCC: atomic built-ins

```
static int global;
void inc() {
  __sync_fetch_and_add(&global, 1);
}
```

Windows

```
static LONG global;
void inc() {
  InterlockedIncrement(&global);
}
```

Atomic Instructions (2)

```
#include <atomic>
std::atomic<int> global(0);
void inc() {
  global++;
}
```

- Specializations for all types that are capable

Self-Deadlocks (1)

Deadlocks: one more dimension in bug-space

- Usually between two threads
- **Self-deadlock**: between one thread

The most obvious self-deadlock

```
std::mutex lock;
...
lock.lock();
lock.lock(); // wait forever
```

Self-Deadlocks (2)

(Only slightly) more intelligent ways to lock the same mutex twice ...

- Calling a callback while holding the lock
  - *What?*
  - *Passing control to untrusted code when critical*??
- Public method uses another public method of the same object
  - → Safer: distinguish between “locked” (public) and “unlocked” (private) methods
  - “locked” may only use “unlocked”

→ Design decision

Working Around Self-Deadlocks: Recursive Mutex

Recursive mutex ...

- Same thread can enter an arbitrary number of times
- Has to exit exactly as many times to release the mutex for *other* threads

The most obvious self-deadlock

```
std::recursive_mutex lock;
...
lock.lock(); // locked for others
lock.lock(); // granted
// ...
lock.unlock();
lock.unlock(); // released for others
```

Overview

FASCHINGBAUER

Condition Variables

Condition Variable

- The most basic communication device
- Everything else can be built around it (and a mutex)
  - Semaphores
  - Events
  - Message queues
  - Promises and futures (→ later)

Best done by example

- `condvar-message-queue.cc`
- `while` instead of `if` → *Spurious Wakeups!*

More Communication: Future

Problem:

- Worker thread calculates *something* in the background
- Somebody waits (synchronizes) for that *something* to become ready
- That *something* will become ready *in the future*

Solution:

- `condvar-future.cc`
    - Manually coded `Future` communication device
    - In terms of good old condition variable and mutex

`std::promise` and `std::future`

Same scenario, but different responsibilities

- Somebody promises to have *something* ready in the future
- Two objects ...
  - `std::promise` is used by producer (the one who promises)
  - `std::future` is used by consumer (who relies on the promise that has been made)

Best done by example

- `promise-future.cc`

Notes