C++ Costless Abstractions

the Compiler View

Proudly made in *Namek* by serge-sans-paille

/me

Serge « sans paille » Guelton

```
$ whoami
sguelton
```

- R&D engineer at QuarksLab on compilation for security
- Associate researcher at Télécom Bretagne

He that started it all

«C++ Is my favorite garbage collected language because it generates so little garbage» – Bjarn Stroustrup

Zero-Cost Abstraction

> «C++ *enables zero-overhead abstraction to get us away from the hardware without adding cost.» – Bjarn Stroustrup*

FACT: There's always a cost.

But the compiler (and the language) pay it in **complexity**.
You don't pay it in **execution time**.

`constexpr` ≠ costless

```
constexpr int fibo(int v) {
  return v < 2 ? v : (fibo(v-1) + fibo(v-2));
}
int main() {
  static_assert(fibo(26) == 121393, "ok");
  // std::cout << fibo(26) << std::endl;
  return 0;
}
```

- Compilation times with `constexpr`: ~0.9s
- Compilation times with**out** `constexpr`: ~0.3s

`constexpr` trivia

Changing Fibo number from **26** to **27**

```
% time clang++ constexpr.cpp -std=c++14
constexpr.cpp:7:17: error: static_assert expression is not an integral co
  static_assert(fibo(27) == 196418, "ok");
                ^~~~~~~~~~~~~~~~~~
constexpr.cpp:2:27: note: constexpr evaluation hit maximum step limit; po
constexpr int fibo(int v) {

}
```

A Brief Word about Clang+LLVM

- Modern compiler infrastructure
- Historical focus on C-like languages
- Uses a typed Internal Representation

Used to illustrate all the following examples

Disclaimer

The results in theses slides are relative to the compiler used (clang++-3.6), the OS (Linux) and the arch (amd64).

Almost nothing is guaranteed by the standard

Functions

Functions are an essential piece of abstraction:

- Give a name to a block of code
- Avoid redundancy
- Abstract with respect to types → overloading

Functions

But a function call implies a performance penality

- **Saving context**
- Jumping (twice)

And they are **over-used in the STL:**

- `std::copy`
- calls `std::__copy_move_a2`
- that calls `__copy_move_a`
- that calls `__copy_move`
- that may calls `__builtin_memmove`

Inlining - Input

```
struct foobar {
  int doit(int a) const { return a+1;}
};

inline int foo(int a) {
  return a+1;
}
int bar(foobar const &fb, int val) {
  return foo(fb.doit(val));
}
```

Inlining - Output

```
define i32 @_Z3barRK6foobari(%struct.foobar* nocapture readnone derefere
  %1 = add nsw i32 %val, 2
    ret i32 %1
    }
```

Inlining

Q: is `inline` useful?

A: not much more than an hint (+ODR)

Q: how to take control?

A: `-mllvm -inline-threshold=225` *or* `-inlinehint-threshold=42`

Q: how to really take control?

A: `__attribute__((always_inline))`

Value semantic

```
void foo(double & f) {
  f += 1.;
}
double bar(double f) {
  return f + 1.;
}
```

Value semantic

```
define void @_Z3fooRd(double* nocapture dereferenceable(8) %f) {
  %1 = load double* %f
  %2 = fadd double %1, 1.000000e+00
  store double %2, double* %f
  ret void
}

define double @_Z3bard(double %f) {
  %1 = fadd double %f, 1.000000e+00
  ret double %1
}
```

Const Ref or Value?

```
static int foo(int const& v) __attribute__((noinline)) { return v; }
static int bar(int v) __attribute__((noinline)) { return v; }

int caller(int v) {
  return foo(v) + bar(v);
}
```

Const Ref or Value?

Compiled with -O2

```
define internal fastcc i32 @_ZL3fooRKi(i32* nocapture readonly dereferenc
  %1 = load i32* %v
  ret i32 %1
}

define internal fastcc i32 @_ZL3bari(i32 %v) {
  ret i32 %v
}
```

Const Ref or Value?

Compiled with -O3

```
define internal fastcc i32 @_ZL3fooRKi(i32 %v.val) {
  ret i32 %v.val
}

define internal fastcc i32 @_ZL3bari(i32 %v) {
  ret i32 %v
}
```

struct const &

```
struct pack {
  unsigned a,b,c;
};

static int foo(pack const& v) __attribute__((noinline)) { return (v.a + v
int caller(pack const& v) {
  return foo(v);
}
```

Passing struct by value

Changing signature!

```
define internal fastcc i32 @_ZL3fooRK4pack(i32 %v.0.0.val, i32 %v.0.1.val
  %1 = add i32 %v.0.1.val, %v.0.0.val
  %2 = add i32 %1, %v.0.2.val
  %3 = udiv i32 %2, 3
  ret i32 %3
}
```

Passing struct by value

Tail call + unboxing

```
_Z6callerRK4pack:
        movl    (%rdi), %eax
        movl    4(%rdi), %esi
        movl    8(%rdi), %edx
        movl    %eax, %edi
    jmp _ZL3fooRK4pack          # TAILCALL

_ZL3fooRK4pack:
        addl    %esi, %edi
        leal    (%rdi,%rdx), %ecx
        movl    $2863311531, %eax
        imulq   %rcx, %rax
        shrq    $33, %rax
        retq
```

Tag dispatching

```
struct tag0 {};
struct tag1 {};
static int foo(int v, tag0) __attribute__((noinline)) { return v + 0; }
static int foo(int v, tag1) __attribute__((noinline)) { return v * 2; }

int caller(int v) {
  return foo(v, tag0{}) + foo(v, tag1{});
}
```

Tag dispatching

No useless argument

```
define internal fastcc i32 @_ZL3fooi4tag0(i32 %v) {
  ret i32 %v
}

define internal fastcc i32 @_ZL3fooi4tag1(i32 %v) {
  %1 = shl nsw i32 %v, 1
  ret i32 %1
}
```

Lambda

```
int (*foo)(int, int) = [](int x, int y) { return x + y; };
int bar(int x, int y) {
  return x + y;
}
```

Lambda

Not different from a regular function?

```
define i32 @_Z3barii(i32 %x, i32 %y) {
  %1 = add nsw i32 %y, %x
  ret i32 %1
}

define internal i32 @"_ZN3$_08__invokeEii"(i32 %x, i32 %y) {
  %1 = add nsw i32 %y, %x
  ret i32 %1
}
```

Lambda + capture

```
auto foo(int val) {
  return [val](int x) { return x + val; };
}
auto bar() {
  return [](int x) { return x * 3 ; };
}
```

Lambda + capture

Not different from its state!

```
define i32 @_Z3fooi(i32 %val) {
  ret i32 %val
}

define void @_Z3barv() {
  ret void
}
```

`struct`, `class` = data

- Use `class` to **box** data with type information
- *Eventually* to associate treatment to data
- Manage the lifetime of data

Boxing values

```
struct A {
  char a_ = 'a';
};
struct B : A {
  char b_ = 'b';
};
B* foo() {
  return new B();
}
```

Boxing values

```
%struct.B = type { %struct.A, i8 }
%struct.A = type { i8 }

; Function Attrs: uwtable
define noalias %struct.B* @_Z3foov() {
  %1 = tail call noalias i8* @_Znwm(i64 2)
  %2 = bitcast i8* %1 to %struct.B*
  %3 = bitcast i8* %1 to i16*
  store i16 0, i16* %3
  store i8 97, i8* %1
  %4 = getelementptr inbounds i8* %1, i64 1
  store i8 98, i8* %4
  ret %struct.B* %2
}
```

Boxing values

```
_Z3foov:
        pushq   %rax
.Ltmp0:
        movl    $2, %edi
        callq   _Znwm
    movw        $0, (%rax)
    movw        $25185, (%rax)
        popq    %rdx
        retq
```

More Boxing/Unboxing

```
struct foo {
  short val_;
  struct {
    short val_;
    struct {
      short val_;
    } nest_;
  } nest_;
};

foo bar(char val) {
  foo a;
  a.val_ = val;
  a.nest_.val_ = val;
  a.nest_.nest_.val_ = val;
  return a;
}
```

More Boxing/Unboxing

Fuse scalars!

```
define i48 @_Z3barc(i8 signext %val) {
  %1 = sext i8 %val to i16
  %2 = zext i16 %1 to i48
  %3 = shl nuw i48 %2, 32
  %4 = shl nuw nsw i48 %2, 16
  %5 = or i48 %4, %2
  %6 = or i48 %5, %3
  ret i48 %6
}
```

Member Functions

```
struct foo {
  int bar() const;
};

int foo_bar(foo const& f) {
  return f.bar();
}
```

Member Functions

Just a function (very Pythonic!)

```
%struct.foo = type { i32 }

define i32 @_Z7foo_barRK3foo(%struct.foo* dereferenceable(4) %f){
  %1 = tail call i32 @_ZNK3foo3barEv(%struct.foo* %f)
  ret i32 %1
}

declare i32 @_ZNK3foo3barEv(%struct.foo*)
```

Default Copy Constructor

```
struct foo {
  int a,b,c,d,e,f,g,h,i,j,k;
};

foo a;

void test(foo b) {
  a = b;
}
```

Default Copy constructor

Memcpy detected!

```
%struct.foo = type { i32, i32, i32, i32, i32, i32, i32, i32, i32, i32, i

@a = global %struct.foo zeroinitializer

; Function Attrs: nounwind uwtable
define void @_Z4test3foo(%struct.foo* byval nocapture readonly %b) {
  %1 = bitcast %struct.foo* %b to i8*
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* bitcast (%struct.foo* @a to i8
  ret void
}
```

Default Copy constructor

Memcpy? Too expansive!

```
struct foo {
  int a,b,c,d;
};

foo a;

void test(foo b) {
  a = b;
}
```

Default Copy constructor

```
 define void @_Z4test3foo(i64 %b.coerce0, i64 %b.coerce1) {
  store i64 %b.coerce0, i64* bitcast (%struct.foo* @a to i64*)
  store i64 %b.coerce1, i64* bitcast (i32* getelementptr inbounds (%struc
  ret void
}
```

Copy Elision

```
struct S {
  double x_, y_, z_;
  S(double x) : x_{x}, y_{x}, z_{x} {}
  S(S const&) = default;
};

S init() {
  return S(1.);
}
S init2() {
  S d(1.);
  return d;
}
```

CopyElision

For free thanks to the representation

```
define void @_Z4initv(%struct.S* noalias nocapture sret %agg.result) {
  %1 = bitcast %struct.S* %agg.result to <2 x double>*
  store <2 x double> <double 1.000000e+00, double 1.000000e+00> <2 x doub
  %2 = getelementptr inbounds %struct.S* %agg.result, i64 0, i32 2
  store double 1.000000e+00, double* %2
  ret void
}
```

CopyElision

Same code with a temporary

```
define void @_Z5init2v(%struct.S* noalias nocapture sret %agg.result) {
  %1 = bitcast %struct.S* %agg.result to <2 x double>*
  store <2 x double> <double 1.000000e+00, double 1.000000e+00> <2 x doub
  %2 = getelementptr inbounds %struct.S* %agg.result, i64 0, i32 2
  store double 1.000000e+00, double* %2
  ret void
```

Virtual method calls

```
struct Interface {
  virtual int doit() = 0;
  virtual ~Interface() {}
};
struct A : Interface {
  int doit() final { return 0; }
};

int foo(Interface& a) {
  return a.doit();
}
int bar(A& a) {
  return a.doit();
}
```

Virtual method calls

Here comes the `vtable`

```
define i32 @_Z3fooR9Interface(%struct.Interface* dereferenceable(8) %a)
  %1 = bitcast %struct.Interface* %a to i32 (%struct.Interface*)***
  %2 = load i32 (%struct.Interface*)*** %1
  %3 = load i32 (%struct.Interface*)** %2
  %4 = tail call i32 %3(%struct.Interface* %a)
  ret i32 %4
}

define i32 @_Z3barR1A(%struct.A* nocapture readnone dereferenceable(8) %a
  ret i32 0
}
```

Virtual method calls

Force virtual call

```
int foobar() {
  A a;
  Interface& b = a;
  return b.doit();
}
```

Virtual method calls

It's devirtualized!

```
define i32 @_Z6foobarv() {
  ret i32 0
}
```

Misc

Various bells and whistles in C++ you don't pay for

Initializer list

```
std::array<int, 16> foo = {0,1,2,3,4,5,6,7,8,9, 0xA, 0xB, 0xC, 0xD, 0xE,
std::vector<int> bar = {0,1,2,3,4,5,6,7,8,9, 0xA, 0xB, 0xC, 0xD, 0xE, 0xF
```

Initializer list

Static initialisation

```
@foo = global %"struct.std::array" { [16 x i32] [i32 0, i32 1, i32 2, i32
```

Initializer list

Runtime initialisation

```
@bar = global %"class.std::vector" zeroinitializer
(...)

__cxx_global_var_init.exit:                       ; preds = %0
  %8 = bitcast i8* %1 to i32*
  store i8* %1, i8** bitcast (%"class.std::vector"* @bar to i8**)
  %9 = getelementptr inbounds i8* %1, i64 64
  store i8* %9, i8** bitcast (i32** getelementptr inbounds (%"class.std:
  store i32 0, i32* %8
  %10 = getelementptr inbounds i8* %1
  %11 = bitcast i8* %10 to i32*
  store i32 1, i32* %11
  %12 = getelementptr inbounds i8* %1
  %13 = bitcast i8* %12 to i32*
  store i32 2, i32* %13
  %14 = getelementptr inbounds i8* %1
  %15 = bitcast i8* %14 to i32*
```

Iterating over a Vector

```
double foo(std::vector<double> const& v) {
  double s = 0.;
  for(auto dat : v)
    s += dat;
  return s;
}
double bar(double const* v, std::size_t n) {
  double s = 0.;
  for(std::size_t i = 0; i < n; ++i)
    s += v[i];
  return s;
}
```

Iterating over a vector

Auto version

```
.lr.ph:                                           ; preds = %.lr.ph.prehe
  %s.01 = phi double [ %8, %.lr.ph ], [ 0.000000e+00, %.lr.ph.preheader ]
  %6 = phi double* [ %9, %.lr.ph ], [ %2, %.lr.ph.preheader ]
  %7 = load double* %6
  %8 = fadd double %s.01, %7
  %9 = getelementptr inbounds double* %6, i64 1
  %10 = icmp eq double* %9, %4
  br i1 %10, label %._crit_edge.loopexit, label %.lr.ph
```

Iterating over a vector

Index version

```
.lr.ph:
  %i.02 = phi i64 [ %5, %.lr.ph ], [ 0, %.lr.ph.preheader ]
  %s.01 = phi double [ %4, %.lr.ph ], [ 0.000000e+00, %.lr.ph.preheader ]
  %2 = getelementptr inbounds double* %v, i64 %i.02
  %3 = load double* %2
  %4 = fadd double %s.01, %3
  %5 = add nuw i64 %i.02, 1
  %exitcond = icmp eq i64 %5, %n
  br i1 %exitcond, label %._crit_edge.loopexit, label %.lr.ph
```

Indexing a vector

```
double foo(std::vector<double> const& data, std::size_t i) {
  return data[i];
}
double bar(double const* data, std::size_t i) {
  return data[i];
}
```

Indexing a vector

Needs an extra adress computation

```
define double @_Z3fooRKSt6vectorIdSaIdEEm(%"class.std::vector"* nocapture
  %1 = getelementptr inbounds %"class.std::vector"* %data, i64 0, i32 0,
  %2 = load double** %1
  %3 = getelementptr inbounds double* %2, i64 %i
  %4 = load double* %3
  ret double %4
}

define double @_Z3barPKdm(double* nocapture readonly %data, i64 %i) {
  %1 = getelementptr inbounds double* %data, i64 %i
  %2 = load double* %1
  ret double %2
}
```

New / Delete

```
double foo(double x, double y) {
  auto * z = new double( x + y);
  auto tmp = *z;
  delete z;
  return tmp;
}
```

New / Delete

No more heap allocation! (what about exceptions?)

```
define double @_Z3foodd(double %x, double %y) {
  %1 = fadd double %x, %y
  ret double %1
```

Except / NoExcept

```
int bar() noexcept;
int foobar();
int foo() {
  try {
    return bar() + foobar();
  }
  catch(...) {
    throw;
  }
}
```

Except / NoExcept

No global registration

```
declare i32 @_Z3barv() #1

declare i32 @_Z6foobarv() #2

attributes #1 = { nounwind "less-precise-fpmad"="false" "no-frame-pointe
attributes #2 = { "less-precise-fpmad"="false" "no-frame-pointer-elim"="
```

Except / NoExcept

Different calling mechanism!

```
define i32 @_Z3foov() #0 {
  %1 = tail call i32 @_Z3barv()
  %2 = invoke i32 @_Z6foobarv()
          to label %3 unwind label %5

; :3                                                ; preds = %0
  %4 = add nsw i32 %2, %1
  ret i32 %4

; :5                                                ; preds = %0
  %6 = landingpad { i8*, i32 } personality i8* bitcast (i32 (...)* @__gxx
          catch i8* null
  %7 = extractvalue { i8*, i32 } %6, 0
  %8 = tail call i8* @__cxa_begin_catch(i8* %7)
  invoke void @__cxa_rethrow()
          to label %15 unwind label %9

; :9                                                ; preds = %5
  %10 = landingpad { i8*, i32 } personality i8* bitcast (i32 (...)* @__gx
          cleanup
  invoke void @__cxa_end_catch()
          to label %11 unwind label %12
```

```
; :12                                                ; preds = %9
  %13 = landingpad { i8*, i32 } personality i8* bitcast (i32 (...)* @__gx
          catch i8* null
  %14 = extractvalue { i8*, i32 } %13, 0
  tail call void @__clang_call_terminate(i8* %14)
  unreachable

; :15                                                ; preds = %5
  unreachable
}
```

Outer Product

```
std::vector<double> outer_product(std::vector<double> const& self, std::v
  std::vector<double> outer(self.size() * other.size());
  for(std::size_t i = 0; i < self.size(); ++i)
    for(std::size_t j = 0; j < other.size(); ++j)
      outer[i * other.size() +j] = self[i] * other[j];
  return outer;
}
```

Outer Product

Vectorized (compiled with `-O3 -march=native`)

```
%wide.load20 = load <4 x double>* %76
%77 = fmul <4 x double> %54, %wide.load
%78 = fmul <4 x double> %59, %wide.load18
%79 = fmul <4 x double> %63, %wide.load19
%80 = fmul <4 x double> %68, %wide.load20
```

Bring Home Message

♥ COMPILERS ♥

They do their job, we do ours

Learn to communicate with them

`info gcc` & `clang --help`

☺ **trust no one & be curious**☺

Verify assembly, verify code

THE END

HIGH-LEVEL THANKS

Joël Falcou and Pierrick Brunet

LOW-LEVEL THANKS

Adrien Guinet and Kévin Szkudlapski

ALL-AROUND THANKS

CppCon & Quarkslab for allowing me to be there!