C++ Object Layout

Johannes Bechberger

April 10, 2019

About This Talk

Based on

- Itaninium C++ ABI
- C++17 Standard
- GCC and Clang source code documentation

Please interrupt me if something sounds fishy. . .

Assembler

```c
int add(int a, int b){
  return a + b;
}
```

```asm
add(int, int):
  push rbp
  # sub rsp, 8
  # due to 128 byte red zone
  mov rbp, rsp
  mov DWORD PTR [rbp-4], edi
  mov DWORD PTR [rbp-8], esi
  mov edx, DWORD PTR [rbp-4]
  mov eax, DWORD PTR [rbp-8]
  add eax, edx
  pop rbp
  ret
```

Assembler

```c
int add(int a, int b){
  return a + b;
}
```

```asm
add(int, int):
  push rbp
  # sub rsp, 8
  # due to 128 byte red zone
  mov rbp, rsp
  mov DWORD PTR [rbp-4], edi
  mov DWORD PTR [rbp-8], esi
  mov edx, DWORD PTR [rbp-4]
  mov eax, DWORD PTR [rbp-8]
  add eax, edx
  pop rbp
  ret

add(int, int): # optimized
  lea eax, [rdi+rsi]
  ret
```

x86 Calling Conventions

The first six integer or pointer arguments are passed in registers (according to the System-V-ABI)

- R**DI**
- R**SI**
- R**DX**
- R**CX**
- R8, R9

the rest on the stack. There are more complex rules for pass-by-value.

Assembler (2)

```c
int sub(int a, int b){
  return add(a, -b);
}
```

```
sub(int, int):
  push rbp
  mov rbp, rsp
  sub rsp, 8
  mov DWORD PTR [rbp-4], edi
  mov DWORD PTR [rbp-8], esi
  mov eax, DWORD PTR [rbp-8]
  neg eax
  mov edx, eax
  mov eax, DWORD PTR [rbp-4]
  mov esi, edx
  mov edi, eax
  call add(int, int)
  leave
  ret
```

Assembler (2)

```c
int sub(int a, int b){
  return add(a, -b);
}
```

```asm
sub(int, int): # optimized
  mov eax, edi
  sub eax, esi
  ret
```

ABI

> *C++ source that is compiled into object files is transformed by the compiler: it arranges objects with* **specific alignment** *and* **in a particular layout**, **mangling names according to a well-defined algorithm**, *has* **specific arrangements for the support of virtual functions**, *etc. These details are* **defined as the compiler Application Binary Interface**, *or ABI. From GCC version 3 onwards the GNU C++ compiler uses an industry-standard C++ ABI,* **the Itanium C++ ABI**.

GCC libstdc++ ABI Policy and Guidelines

C++ ABIs on different Platforms

Unix	Itanium ABI, primary ABI used by Clang and GCC, based on System V
ARM	Modified Itanium ABI, modified member function pointers, constructors and destructors return `this`, . . .
iOS	Partial implementation of the ARM ABI
MIPS	Modified version of Itanium, member function pointer are adjusted
WebAssembly	Modified Version of Itanium, adjustment in direction of ARM
Microsoft	"Only scattered and incomplete official documentation exists."

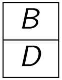
https://clang.llvm.org/doxygen/classclang_1_1TargetCXXABI.html

Object Layout

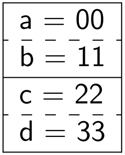
```
struct B  {
  char a = 00;
  char b = 11;
};
struct D : B  {
  char c = 22;
  char d = 33;
};

  D *y = new D();
  y->d; //?
```

Sub-objects of D



Memory layout of y

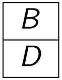


Object Layout

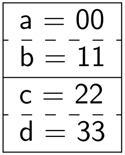
```
struct B  {
  char a = 00;
  char b = 11;
};
struct D : B  {
  char c = 22;
  char d = 33;
};
```

```
  D *y = new D();
  y->d; //?
  // *(y + offset(D::d)) == *(y + 3)
```

Sub-objects of D



Memory layout of y



Object Layout

```
struct B  {
  char a = 00;
  char b = 11;
};
struct D : B  {
  char c = 22;
  char d = 33;
};
```

```
  D *y = new D();
  y->d; //?
  // *(y + offset(D::d)) == *(y + 3)
  reinterpret_cast<B*>(y) == y; //?
```

Sub-objects of D

B
D

Memory layout of y

a = 00
b = 11
c = 22
d = 33

Object Layout

```
struct B  {
  char a = 00;
  char b = 11;
};
struct D : B  {
  char c = 22;
  char d = 33;
};
```

Sub-objects of D

B
D

Memory layout of y

a = 00
b = 11
c = 22
d = 33

```
D *y = new D();
y->d; //?
// *(y + offset(D::d)) == *(y + 3)
reinterpret_cast<B*>(y) == y; //?  // yes
```

See on https://godbolt.org/z/n0eaIG

Be Aware

Non-static data members of a (non-union) class with the same access control (Clause 14) are allocated so that **later members have higher addresses within a class object***. The order of allocation of non-static data members with different access control is unspecified (Clause 14).* **Implementation alignment requirements might cause two adjacent members not to be allocated immediately after each other** *[. . .].*

Source: C++ 17 Standard, N4659

Statically Bound Methods

```
struct B {
  char a = 00; char b = 11;

  char get(){  return this->a;  }
};

struct D : B {
  char c = 22; char d = 33;

  char get(){  return this->c;  }
};

D d;
((B)d).get_b() // ?
```

Statically Bound Methods

```
struct B {
  char a = 00; char b = 11;

  char get(){  return this->a;  }
};

struct D : B {
  char c = 22; char d = 33;

  char get(){  return this->c;  }
};

D d;
((B)d).get_b() // ?  // call B::get
```

See on https://godbolt.org/z/PWovRj

VTables

```
struct B {
  char a = 11;
  virtual void f();
  virtual void g();};
struct D : B {
  char b = 22;
  virtual void f();
  virtual void h();};
```

VTables

```
struct B {
  char a = 11;
  virtual void f();
  virtual void g();};
struct D : B {
  char b = 22;
  virtual void f();
  virtual void h();};
void call(B b){
  b.f(); // ?
```

VTables

```
struct B {
  char a = 11;
  virtual void f();
  virtual void g();};
struct D : B {
  char b = 22;
  virtual void f();
  virtual void h();};
void call(B b){
  b.f(); // ?  // B::f
}
```

VTables

```
struct B {
  char a = 11;
  virtual void f();
  virtual void g();};
struct D : B {
  char b = 22;
  virtual void f();
  virtual void h();};
void call(B b){
  b.f(); // ?  // B::f
}
void call_ptr(B *b){
  b->f(); // ?
```

VTables

```
struct B {
  char a = 11;
  virtual void f();
  virtual void g();};
struct D : B {
  char b = 22;
  virtual void f();
  virtual void h();};
void call(B b){
  b.f(); // ?  // B::f
}
void call_ptr(B *b){
  b->f(); // ?  // B::f or D::f
}
```

VTable of B

$B :: f$
$B :: g$

VTable of D

$D :: f$
$B :: g$
$D :: h$

Instance of D

`&D-vtable`
a = 11
b = 22

VTables

```
struct B {
  char a = 11;
  virtual void f();
  virtual void g();};
struct D : B {
  char b = 22;
  virtual void f();
  virtual void h();};
void call(B b){
  b.f(); // ?  // B::f
}
void call_ptr(B *b){
  b->f(); // ?  // B::f or D::f
}
```

```
B *b = ...;
b->f(); // ?
```

VTable of B

$B :: f$
$B :: g$

VTable of D

$D :: f$
$B :: g$
$D :: h$

Instance of D

`&D-vtable`
a = 11
b = 22

VTables

```
struct B {
  char a = 11;
  virtual void f();
  virtual void g();};
struct D : B {
  char b = 22;
  virtual void f();
  virtual void h();};
void call(B b){
  b.f(); // ?  // B::f
}
void call_ptr(B *b){
  b->f(); // ?  // B::f or D::f
}
```

VTable of B

$B :: f$
$B :: g$

VTable of D

$D :: f$
$B :: g$
$D :: h$

Instance of D

`&D-vtable`
a = 11
b = 22

```
B *b = ...;
b->f(); // ? // (*(*(b + offset(vptr)) + offset(f)))();
```

See on https://godbolt.org/z/Tavc3W

this Pointer

```
struct C {
  char x;
  virtual void m(int y){
    this.x = y;
    this.m(y);
  }
}
```

this Pointer

```
struct C {
  char x;
  virtual void m(int y){
    this.x = y;
    this.m(y);
  }
}
```

```
struct C {
  vtable *vptr;
  char x;
}
void m(C *this, char y){
  *(this + offset(C::x)) = y;
  (*(*(this + offset(vptr))
     + offset(C::m))(this, y);
}
```

Downcasts

```
struct B {char a;};
struct D : B {int b;};

B b;
((D*)&b); // ?
```

Downcasts

```
struct B {char a;};
struct D : B {int b;};

B b;
((D*)&b); // ? // null code
```

Casts of Values

```
struct B {
  char a = 1; char b = 2; char c = 3;
  virtual char g();
};

struct D : B {
  char c = 4;
  virtual char g();
};

D d; B b;
((B)d); // ?
```

Casts of Values

```
struct B {
  char a = 1; char b = 2; char c = 3;
  virtual char g();
};

struct D : B {
  char c = 4;
  virtual char g();
};

D d; B b;
((B)d); // ?  // creates copy
```

Casts of Values

```
struct B {
  char a = 1; char b = 2; char c = 3;
  virtual char g();
};

struct D : B {
  char c = 4;
  virtual char g();
};

D d; B b;
((B)d); // ?  // creates copy
((D)b); // ?
```

Casts of Values

```
struct B {
  char a = 1; char b = 2; char c = 3;
  virtual char g();
};

struct D : B {
  char c = 4;
  virtual char g();
};

D d; B b;
((B)d); // ?  // creates copy
((D)b); // ?  // not allowed
```

Casts of Values

```
struct B {
  char a = 1; char b = 2; char c = 3;
  virtual char g();
};

struct D : B {
  char c = 4;
  virtual char g();
};

D d; B b;
((B)d); // ?  // creates copy
((D)b); // ?  // not allowed
```

See on https://godbolt.org/z/somhVA

Calling Dynamic Methods in the Constructor

```
struct B {
  B(){
    g();
    // how is this bound and why?
  }
  char a;
  virtual void g(){
    print(1);
  }
};
```

Calling Dynamic Methods in the Constructor

```
struct B {
  B(){
    g(); // statically bound
    // sets vptr afterwards
  }
  char a;
  virtual void g(){
    print(1);
  }
};
```

See on https://godbolt.org/z/HP8RgV

this and Upcasts

```
struct B { char a = 11;
  void f(); };
struct D : B { char b = 22;
  virtual void g(); };

D *d;
(B*)d; // ?
```

this and Upcasts

```
struct B { char a = 11;
  void f(); };
struct D : B { char b = 22;
  virtual void g(); };

D *d;
(B*)d; // ?  // not null code
```

See on https://godbolt.org/z/2jb2VJ

Multiple Inheritance

> *For each distinct occurrence of a non-virtual base class in the class lattice of the most derived class, the most derived object (4.5) shall contain a corresponding distinct base class subobject of that type.*
>
> *For each distinct base class that is specified virtual, the most derived object shall contain a single base class subobject of that type.*

Source: C++ 17 Standard, N4659

Non-virtual Multiple Inheritance

```
struct L { char l = 00;
  virtual void f(){} };
struct A : L { char a = 11;
  virtual void f(){} };
struct B : L { char b = 22; };
struct C : A, B { char c = 33; };

C c;
c.l; // ?
```

Non-virtual Multiple Inheritance

```
struct L { char l = 00;
  virtual void f(){} };
struct A : L { char a = 11;
  virtual void f(){} };
struct B : L { char b = 22; };
struct C : A, B { char c = 33; };

C c;
c.l; // ?  // ambiguous
// C contains two Ls
```

Non-virtual Multiple Inheritance

```
struct L { char l = 00;
  virtual void f(){} };
struct A : L { char a = 11;
  virtual void f(){} };
struct B : L { char b = 22; };
struct C : A, B { char c = 33; };

C c;
c.l; // ?  // ambiguous
// C contains two Ls
((B)c)).l; // ?
```

Non-virtual Multiple Inheritance

```
struct L { char l = 00;
  virtual void f(){} };
struct A : L { char a = 11;
  virtual void f(){} };
struct B : L { char b = 22; };
struct C : A, B { char c = 33; };

C c;
c.l; // ?  // ambiguous
// C contains two Ls
((B)c)).l; // ?  // ok
```

See on https://godbolt.org/z/3yy9C2

Sub-objects of C

$C \cdot A \cdot L$
$C \cdot A$
$C \cdot B \cdot L$
$C \cdot B$
C

Multiple Inheritance and Order

The order of derivation is not significant except as specified by the semantics of initialization by constructor (15.6.2), cleanup (15.4), and storage layout.

Source: C++ 17 Standard, N4659

Virtual Multiple Inheritance

```
struct L { char l = 00;
  virtual void f(){} };
struct A : virtual L { char a = 11; };
struct B : virtual L { char b = 22; };
struct C : A, B { char c = 33; };
C c;
c.l; // ?
```

Virtual Multiple Inheritance

```
struct L { char l = 00;
  virtual void f(){} };
struct A : virtual L { char a = 11; };
struct B : virtual L { char b = 22; };
struct C : A, B { char c = 33; };
C c;
c.l; // ?  // ok
c.f(); // ?
```

Virtual Multiple Inheritance

Sub-objects of *C*

A
B
C
V

```
struct L { char l = 00;
  virtual void f(){} };
struct A : virtual L { char a = 11; };
struct B : virtual L { char b = 22; };
struct C : A, B { char c = 33; };
C c;
c.l; // ?  // ok
c.f(); // ?
// *(c + offset(vptr))[0][0](c + *(...)[0][1])
```

Memory layout of *A* sub-object

Offset to *L*
a = 11

VTable of *C*

C::f	*L* - *C*

See on https://godbolt.org/z/uQOY4K

Thunks and Offsets

Two ways to implement method calls that include the upcast:

Offsets ► cast on every call to a virtual method

Thunks and Offsets

Two ways to implement method calls that include the upcast:

Offsets	▶ cast on every call to a virtual method
Thunks	▶ ignore offset ▶ create small method (thunk) that casts the value and calls the method ▶ faster in case of non-virtual or non-multiple inheritance case ▶ see e.g. D

Be aware

The order in which the base class subobjects are allocated in the most derived object is unspecified.

Source: C++ 17 Standard, N4659

the whole “virtual call” mechanism is also undefined, this is loosely defined by the ABI

Demos

- using the object layout for fun and profit
- assuming a fixed compiler, ABI and object layout
- Ideas
  - Become example
  - Sub-object pointer example
  - RTTI example
- Examples are available at GitHub

Become Example (by Andreas Fried)

```
struct Animal {
  const std::string name;  virtual void make_a_noise() = 0;
};

struct Dog : public Animal {
  virtual void make_a_noise() {
    std::cout << name << ": Wuff!" << std::endl;
  }
};

struct Cat : public Animal {
  virtual void make_a_noise() {
    std::cout << name << ": Miau!" << std::endl;
  }
};

template <class T> void become(void *obj) {
  T tmp;
  memcpy(obj, &tmp, sizeof(void*));
}
```

VBase Pointer Example

```
struct A {
  virtual void print(){
    std::cout << "I belong to A" << std::endl;
  }};

struct B : public virtual A {};

struct NotA {
  virtual void not_print(){
    std::cout << "Who's A?" << std::endl;
  }};

template <typename T1, typename T2>
void change_vbase_ptr(T1 *obj, T2 *other){
  void** vtbl = ((void***)obj)[0];
  size_t * aligned_vtbl = (size_t *) ((size_t) vtbl &~(4096-1));
  mprotect(aligned_vtbl, 1024, PROT_WRITE|PROT_EXEC|PROT_READ );
  ((size_t*)vtbl)[-3] = (size_t)other - (size_t)obj;
}
```

Before the VTable

Offset to virtual base class 1
. . .
Offset to virtual base class n
Offset to top
Pointer to RTTI

Offset to top: *find the top of the object from any base subobject with a virtual table pointer* (Itanium ABI)

RTTI Example

```
class A {
  virtual void blub(){}
};

template <typename T>
void change_name(T* obj, const std::string &name) {
  void **vtbl = ((void ***) obj)[0];
  void **type_info = (void **) vtbl[-1];
  make_writable(type_info);
  const char *str = (new std::string(name))->c_str();
  ((char **) type_info)[1] = (char *) str;
}

int main(){
  A a;
  std::cout << typeid(a).name() << std::endl;
  change_name(&a, "Hello World!");
  std::cout << typeid(a).name() << std::endl;
}
```

And Now: Something Completely Different

The world of casts

What is a Cast?

An explicit type conversion can be expressed using functional notation (8.2.3), a type conversion operator (`dynamic_cast`*,* `static_cast`*,* `reinterpret_cast`*,* `const_cast`*), or the cast notation*

Source: C++ 17 Standard, N4659

Grammar Excerpt

```
multiplicative-expression:
        pm-expression
        ...
pm-expression:
        cast-expression
        ...
cast-expression:
        unary-expression
        ( type-id ) cast-expression
postfix-expression:
        dynamic_cast < type-id > ( expression )
        static_cast < type-id > ( expression )
        reinterpret_cast < type-id > ( expression )
        const_cast < type-id > ( expression )
        ...
```

Source: C++ 17 Standard, N4659

What is a Cast?

Any type conversion not mentioned below and not explicitly defined by the user (15.3) is ill-formed.

- *a* `const_cast`,
- *a* `static_cast`,
- *a* `static_cast` *followed by a* `const_cast`,
- *a* `reinterpret_cast`, *or*
- *a* `reinterpret_cast` *followed by a* `const_cast`

. . .
If a conversion can be interpreted in more than one of the ways listed above, the interpretation that appears first in the list is used, even if a cast resulting from that interpretation is ill-formed.

Source: C++ 17 Standard, N4659

Static Cast

```
static_cast<cv D>(b) // when is this okay?
```

Static Cast

```
static_cast<cv D>(b) // when is this okay?
```

Iff

- D is a class
- $D \leq B$
- B is not a virtual base class of D
- there exists a valid standard conversion of B* to D*
- if B is more or equal const than D

$\Rightarrow$ null-code

Source: C++ 17 Standard, N4659

Static Cast

```
struct D : public B { };
D d;
B &br = d;
static_cast<D&>(br);
```

See on https://godbolt.org/z/gcGd3k

Source: C++ 17 Standard, N4659

Static Cast: Run-Time Checks

```
#include <iostream>

struct A { virtual void f(){} };
struct D : A { int a = 3; int b = 4;};
struct E : A { char a = 4; };

void h(A *a) {
  D* b = static_cast<D*>(a);
}
```

Static Cast: Run-Time Checks

```
#include <iostream>

struct A { virtual void f(){} };
struct D : A { int a = 3; int b = 4;};
struct E : A { char a = 4; };

void h(A *a) {
  D* b = static_cast<D*>(a);
}
```

No check is done at run-time

Reinterpret Cast

- works for pointers and references
- including standard array and function pointer conversions
- "The mapping performed by `reinterpret_cast` might, or might not, produce a representation different from the original value"
- "A pointer can be explicitly converted to any integral type large enough to hold it."
- " The mapping function is implementation-defined."
- cannot cast away constness

Source: C++ 17 Standard, N4659

Const Cast

```
const_cast<cv T1>(t2) // when is this okay?
```

Const Cast

```
const_cast<cv T1>(t2) // when is this okay?
```

Iff

- T is similar to t2
- lvalue-to-rvalue, array-to-pointer, and function-to-pointer (7.3) standard conversions applied

Dynamic Cast

```
dynamic_cast<cv B>(D) // when is this okay?
```

Dynamic Cast

```
dynamic_cast<cv B>(D) // when is this okay?
```

- works for pointers or references
- requires that the B subobject in D is unique
- cannot cast away constness
- if used in constructors: —

Dynamic Cast

```
dynamic_cast<cv B>(D) // when is this okay?
```

- works for pointers or references
- requires that the B subobject in D is unique
- cannot cast away constness
- if used in constructors: —— goes from class type upwards
- the only one that checks at run-time
  (either exception or null as a return value on error)

Dynamic Cast

```
dynamic_cast<void*>(b); // ?
```

Dynamic Cast

```
dynamic_cast<void*>(b); // ?
```

“If T is ‘pointer to cv void’, then the result is a pointer to the most derived object pointed to by v.”

Dynamic Cast

```
class A { virtual void f(); };
class B { virtual void g(); };
class D : public virtual A, private B { };
void g() {
  D d;
  B* bp = (B*)&d;  // cast needed to break protection
  A* ap = &d;      // ?
```

Dynamic Cast

```
class A { virtual void f(); };
class B { virtual void g(); };
class D : public virtual A, private B { };
void g() {
  D d;
  B* bp = (B*)&d;  // cast needed to break protection
  A* ap = &d;      // ? // public derivation
                        // no cast needed
  D& dr = dynamic_cast<D&>(*bp); // ?
```

Dynamic Cast

```
class A { virtual void f(); };
class B { virtual void g(); };
class D : public virtual A, private B { };
void g() {
  D d;
  B* bp = (B*)&d;  // cast needed to break protection
  A* ap = &d;      // ? // public derivation
                        // no cast needed
  D& dr = dynamic_cast<D&>(*bp); // ? // fails
  ap = dynamic_cast<A*>(bp);     // ?
```

Dynamic Cast

```
class A { virtual void f(); };
class B { virtual void g(); };
class D : public virtual A, private B { };
void g() {
  D d;
  B* bp = (B*)&d;  // cast needed to break protection
  A* ap = &d;      // ? // public derivation
                        // no cast needed
  D& dr = dynamic_cast<D&>(*bp); // ? // fails
  ap = dynamic_cast<A*>(bp);     // ? // fails
  bp = dynamic_cast<B*>(ap);     // ?
```

Dynamic Cast

```
class A { virtual void f(); };
class B { virtual void g(); };
class D : public virtual A, private B { };
void g() {
  D d;
  B* bp = (B*)&d;  // cast needed to break protection
  A* ap = &d;      // ? // public derivation
                        // no cast needed
  D& dr = dynamic_cast<D&>(*bp); // ? // fails
  ap = dynamic_cast<A*>(bp);     // ? // fails
  bp = dynamic_cast<B*>(ap);     // ? // fails
  ap = dynamic_cast<A*>(&d);     // ?
```

Dynamic Cast

```
class A { virtual void f(); };
class B { virtual void g(); };
class D : public virtual A, private B { };
void g() {
  D d;
  B* bp = (B*)&d;  // cast needed to break protection
  A* ap = &d;      // ? // public derivation
                        // no cast needed
  D& dr = dynamic_cast<D&>(*bp); // ? // fails
  ap = dynamic_cast<A*>(bp);     // ? // fails
  bp = dynamic_cast<B*>(ap);     // ? // fails
  ap = dynamic_cast<A*>(&d);     // ? // succeeds
  bp = dynamic_cast<B*>(&d);     // ?
```

Dynamic Cast

```
class A { virtual void f(); };
class B { virtual void g(); };
class D : public virtual A, private B { };
void g() {
  D d;
  B* bp = (B*)&d;  // cast needed to break protection
  A* ap = &d;      // ? // public derivation
                        // no cast needed
  D& dr = dynamic_cast<D&>(*bp); // ? // fails
  ap = dynamic_cast<A*>(bp);     // ? // fails
  bp = dynamic_cast<B*>(ap);     // ? // fails
  ap = dynamic_cast<A*>(&d);     // ? // succeeds
  bp = dynamic_cast<B*>(&d);     // ? // ill-formed
                                 // (not a runtime check)
}
```

See on https://godbolt.org/z/T2HiYE

Source: C++ 17 Standard, N4659

Dynamic Cast

```
class A { virtual void f(); };
class B { virtual void g(); };
class D : public virtual A, private B { };
class E : public D, public B { };
class F : public E, public D { };
void h() {
  F f;
  A* ap = &f; // ?
```

Dynamic Cast

```
class A { virtual void f(); };
class B { virtual void g(); };
class D : public virtual A, private B { };
class E : public D, public B { };
class F : public E, public D { };
void h() {
  F f;
  A* ap = &f; // ? // succeeds: finds unique A
  D* dp = dynamic_cast<D*>(ap); // ?
```

Dynamic Cast

```
class A { virtual void f(); };
class B { virtual void g(); };
class D : public virtual A, private B { };
class E : public D, public B { };
class F : public E, public D { };
void h() {
  F f;
  A* ap = &f; // ? // succeeds: finds unique A
  D* dp = dynamic_cast<D*>(ap); // ?
     // fails: yields null; f has two D subobjects
  E* ep = (E*)ap; // ?
```

Dynamic Cast

```
class A { virtual void f(); };
class B { virtual void g(); };
class D : public virtual A, private B { };
class E : public D, public B { };
class F : public E, public D { };
void h() {
  F f;
  A* ap = &f; // ? // succeeds: finds unique A
  D* dp = dynamic_cast<D*>(ap); // ?
     // fails: yields null; f has two D subobjects
  E* ep = (E*)ap; // ?
     // ill-formed: cast from virtual base
  E* ep1 = dynamic_cast<E*>(ap); // ?
```

Dynamic Cast

```
class A { virtual void f(); };
class B { virtual void g(); };
class D : public virtual A, private B { };
class E : public D, public B { };
class F : public E, public D { };
void h() {
  F f;
  A* ap = &f; // ? // succeeds: finds unique A
  D* dp = dynamic_cast<D*>(ap); // ?
     // fails: yields null; f has two D subobjects
  E* ep = (E*)ap; // ?
     // ill-formed: cast from virtual base
  E* ep1 = dynamic_cast<E*>(ap); // ? // succeeds
}
```

See on https://godbolt.org/z/0M06nU

Source: C++ 17 Standard, N4659

Dynamic Cast: Implementation

Type information for a class with multiple and/or virtual bases, excerpt from GCC libstdc++-v3:

```
class __vmi_class_type_info : public __class_type_info
{
  const char *__name; // from std::type_info
  unsigned int __flags;
  unsigned int __base_count;
  __base_class_type_info  __base_info[1];
  ...
  enum __flags_masks
  {
    __non_diamond_repeat_mask = 0x1,
    __diamond_shaped_mask = 0x2,
    __flags_unknown_mask = 0x10
  };
  ...
};
```

Dynamic Cast: Implementation

Any ideas?

Dynamic Cast: Implementation

Simplest Depth-first search in DAG

Dynamic Cast: Implementation

Simplest Depth-first search in DAG. Uses?

Dynamic Cast: Implementation

Simplest Depth-first search in DAG (clang and gcc)

Dynamic Cast: Implementation

Simplest	Depth-first search in DAG (clang and gcc)
Better	Use prime ids to check that a class is a sub-class of another, in each step (see Fast Dynamic Casting (2006)) Not used as compatibility makes changes difficult

When To Use Static Casts Instead Of Dynamic Casts

- if the provided `static_cast` is really slow
- ...and the cast is performance critical
- ...and you have other means to guarantee that a downcast will succeed
- ...and there is no virtual inheritance involved
- ...and you added a prominent disclaimer

See C++ Core Guidelines

Conclusion

- Reading the standard helps
- Using a simple class hierarchy without multiple inheritance too
- There are many traps to fall in