PTV GROUP
the mind of movement

Curious C++

From the daily weirdness to the shady corners of the language

Achim Guckenberger
PTV Group

C++ User Group Karlsruhe
11.09.2019

Comparisons for equality

PTV GROUP
the mind of movement

Let's start simple…

Assume we have

```
template <class T>
bool Equal(T val1, T val2) {
  return val1 == val2;
}
```

Question: In which cases can the following happen?

```
Equal(someValue, someValue) == false
```

Let's start simple...

Cases when this can happen:

1. If `T` has an overloaded `operator==`
   ➔ Result depends on implementation
2. If `T` is a floating point type (`float, double, long double`) and the value is `NaN`.
   Reason:

   ```
   NaN != NaN
   ```

Could be exploited to check for `NaN`:

- `someDouble!=someDouble`:
  Can break with `-ffast-math` (ignores IEEE specification)!
- Use `std::isnan()`! Simpler, clearer and safer.

Mystic colons

Mystic colons

Question: What does the following mean:

```
struct S
{
  int someValue : 1;
};
```

Answer: It is a bitfield!

Purpose: Allows to pack several variables into a byte:

```
struct S {
 int value1 : 3;
 bool value2 : 1;
 short value3 : 4;
};
```

Mystic colons

Uses:

- Memory space optimization (but: slow access!).
  Nowadays: Almost never useful in desktop environments.
- Talking to microcontrollers registers
- Lock-free multithreading: Atomic compare & exchange of whole struct

Problems:

- Packing is implementation defined!
- Cannot take address or reference
- Slow

Mystic colons

Some additional weirdness:

```
struct S {
  int value1 : 1024; // Only sizeof(int) is usable
  bool : 3; // 3 padding bits
  int value2 : 2;
  char : 0; // 0: Start new "pack"
  unsigned value3 : 4;
};
```

Return value optimization

Return value optimization

Consider:

```
SomeType Func() {
  return SomeType();
}
SomeType t = Func();
```

➔ In principle: 1 default constructor + 2 copy constructor calls.

Copy elision:

- Before C++17: One or both **may** be omitted
- With C++17: Both **must** be omitted.

Return value optimization

But: Only for „prvalues" it is guaranteed.

Hence:

```
SomeType Func() {
  SomeType s;
  return s;
}
```

➔ Copy constructor **can** be omitted, but does not have to

Return value optimization

What about this?

```
SomeType Func() {
  SomeType s;
  return std::move(s);
}
```

➔ Prevents return value optimization (move constructor must be called)

Neat code… But is there an error?

```
unsigned GetBucketOfNumber(unsigned number, std::vector<unsigned> const & sortedBuckets)
{
  for (size_t bucketIdx = 0; bucketIdx < sortedBuckets.size(); ++bucketIdx) {
    if (number < sortedBuckets[bucketIdx])
      return bucketIdx;
  }
  return sortedBuckets.size();
}

int main()
{
  std::vector<unsigned> const DATA_BINS {
    0001,
    0010,
    0100,
    1000
  };

  std::cout << GetBucketOfNumber(70, DATA_BINS) << std::endl;
}
```

Neat code… But is there an error?

```
unsigned GetBucketOfNumber(unsigned number, std::vector<unsigned> const & sortedBuckets)
{
  for (size_t bucketIdx = 0; bucketIdx < sortedBuckets.size(); ++bucketIdx) {
    if (number < sortedBuckets[bucketIdx])
      return bucketIdx;
  }
  return sortedBuckets.size();
}

int main()
{
  std::vector<unsigned> const DATA_BINS {
    0001, // == 1
    0010, // == 8 (!!)
    0100, // == 64 (!!)
    1000  // == 1000
  };

  std::cout << GetBucketOfNumber(70, DATA_BINS) << std::endl; // Prints 3, not 2!
}
```


PTV GROUP
the mind of movement

Signed vs unsigned vs … nothingness

Some more fun: Which functions get called?

```
void func(int) { cout << "int" << endl; }
void func(signed int) { cout << "signed int" << endl; }
void func(unsigned int) { cout << "unsigned int" << endl; }


void func(char) { cout << "char" << endl; }
void func(signed char) { cout << "signed char" << endl; }
void func(unsigned char) { cout << "unsigned char" << endl; }

int main()
{
  int i = 0;
  func(i);

  char c = 1;
  func(c);
}
```

Some more fun: Which functions get called?

```
// error C2084: function 'void func(int)' already has a body: int == signed int
void func(int) { cout << "int" << endl; }
// void func(signed int) { cout << "signed int" << endl; }
void func(unsigned int) { cout << "unsigned int" << endl; }

// OK: char != signed char != unsigned char
void func(char) { cout << "char" << endl; }
void func(signed char) { cout << "signed char" << endl; }
void func(unsigned char) { cout << "unsigned char" << endl; }

int main()
{
  int i = 0;
  func(i);

  char c = 1;
  func(c);
}
```


PTV GROUP
the mind of movement

( )

Parentheses everywhere! ((1))

Is there a difference between the following?

```
S * s = new S;   // (1)
S * s = new S(); // (2)
```

Answer: Depends on S!

- No user-provided constructor:
  ➔ (1): Uninitialized (except for default member initializers)
  ➔ (2): Zero-initialized
- User-provided constructor
  ➔ Constructor gets called.

Parentheses everywhere! ((2))

Question: What is the return type?

```
struct S {
  int & GetVal();
  int val;
};

[](S & s) { return s.GetVal(); };                    // returns int
```

Parentheses everywhere! ((2))

Question: What is the return type?

```
struct S {
  int & GetVal();
  int val;
};
[](S & s) -> auto { return s.GetVal(); };          // returns int
```

Parentheses everywhere! ((2))

Question: What is the return type?

```
struct S {
  int & GetVal();
  int val;
};
[](S & s) -> auto { return s.GetVal(); };           // returns int
[](S & s) -> decltype(auto) { return s.GetVal(); }; // returns int &
[](S & s) -> decltype(auto) { return s.val; };      // returns int
[](S & s) -> decltype(auto) { return (s.val); };    // returns int &
```

Notes:

- `auto` is never a reference.
- `decltype(auto)` keeps & and &&. `template <class T> decltype(auto) f(T & t) { return t.f(); }`
- `s.val`: class member-access expression. `(s.val)`: lvalue expression.
  `decltype(x) != decltype((x))`

& = | ~ ! ^

Alternative tokens

Problem: My keyboard does not have any of the following characters:

& = | ~ ! ^

Solution: Use alternative tokens!

<table>
<tr><td>&&</td><td>and</td></tr>
<tr><td>&=</td><td>and_eq</td></tr>
<tr><td>&</td><td>bitand</td></tr>
<tr><td>|</td><td>bitor</td></tr>
<tr><td>~</td><td>compl</td></tr>
<tr><td>!</td><td>not</td></tr>
<tr><td>!=</td><td>not_eq</td></tr>
<tr><td>||</td><td>or</td></tr>
<tr><td>|=</td><td>or_eq</td></tr>
<tr><td>^</td><td>xor</td></tr>
<tr><td>^=</td><td>xor_eq</td></tr>
</table>

Alternative tokens

```cpp
class Foo {
public:
  Foo();
  ~Foo();
  Foo(Foo const & foo);
  Foo(Foo && foo);
};
```

Alternative tokens

```
class Foo {
public:
  Foo();
  compl Foo();
  Foo(Foo const bitand foo);
  Foo(Foo and foo);
};
```

The many addresses of objects

The many addresses of objects

Non-polymorphic inheritance:

```
struct A {
    int val1;
};

struct B : public A {
    int val2;
};

B b;
B * pB = &b;
A * pA = pB;

cout << "pB = 0x" << hex << pB << endl;
cout << "pA = 0x" << hex << pA << endl;
```

The many addresses of objects

Non-polymorphic inheritance:

```
struct A {
   int val1;
};

struct B : public A {
   int val2;
};

B b;
B * pB = &b;
A * pA = pB;

cout << "pB = 0x" << hex << pB << endl; // pB = 0xF7B8
cout << "pA = 0x" << hex << pA << endl; // pA = 0xF7B8
```

➔ Addresses are equal

The many addresses of objects

Polymorphic case:

```
struct A {
   virtual ~A() = default;
   int val1;
};

struct B : public A {
   int val2;
};

B b;
B * pB = &b;
A * pA = pB;

cout << "pB = 0x" << hex << pB << endl;
cout << "pA = 0x" << hex << pA << endl;
```

The many addresses of objects

Polymorphic case:

```
struct A {
   virtual ~A() = default;
   int val1;
};

struct B : public A {
   int val2;
};

B b;
B * pB = &b;
A * pA = pB;

cout << "pB = 0x" << hex << pB << endl; // pB = 0xF7B8
cout << "pA = 0x" << hex << pA << endl; // pA = 0xF7B8
```

➔ Addresses are equal

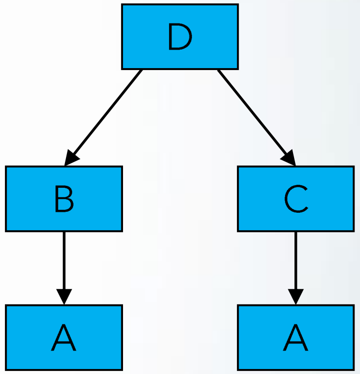
The many addresses of objects

Rather known: Multiple inheritance ➔ multiple object parts ➔ multiple addresses

```
struct A {
   virtual ~A() = default;
   int val1;
};

struct B : public A            { int val2; };
struct C : public A            { int val3; };
struct D : public B, public C { int val4; };

D d;
D * pD = &d;
B * pB = pD;
C * pC = pD;
A * pA_B = pB;
A * pA_C = pC;
```



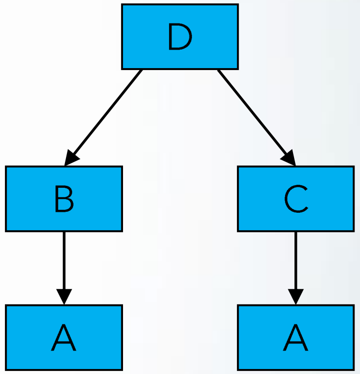
The many addresses of objects

Rather known: Multiple inheritance ➔ multiple object parts ➔ multiple addresses

```
struct A {
    virtual ~A() = default;
    int val1;
};

struct B : public A           { int val2; };
struct C : public A           { int val3; };
struct D : public B, public C { int val4; };

D d;
D * pD = &d;    // 0xF8C8
B * pB = pD;    // 0xF8C8
C * pC = pD;
A * pA_B = pB; // 0xF8C8
A * pA_C = pC;
```



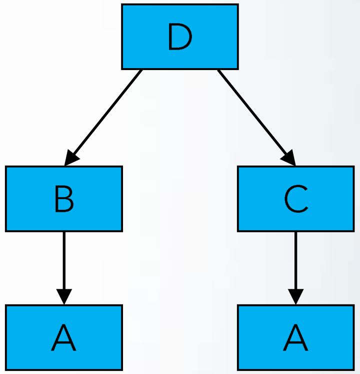
The many addresses of objects

Rather known: Multiple inheritance ➔ multiple object parts ➔ multiple addresses

```
struct A {
   virtual ~A() = default;
   int val1;
};

struct B : public A           { int val2; };
struct C : public A           { int val3; };
struct D : public B, public C { int val4; };

D d;
D * pD = &d;    // 0xF8C8
B * pB = pD;    // 0xF8C8
C * pC = pD;    // 0xF8E0 (!)
A * pA_B = pB; // 0xF8C8
A * pA_C = pC; // 0xF8E0 (!)
```



➔ The parts B and C have distinct addresses!
(Reason: Multiple vtables)

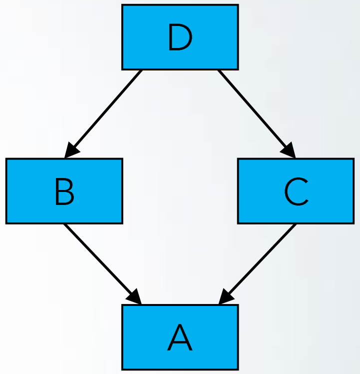
The many addresses of objects

Virtual inheritance (➔ only one part A):

```
struct A {
   virtual ~A() = default;
   int val1;
};

struct B : virtual public A   { int val2; };
struct C : virtual public A   { int val3; };
struct D : public B, public C { int val4; };

D d;
D * pD = &d;
B * pB = pD;
C * pC = pD;
A * pA_B = pB;
A * pA_C = pC;
```



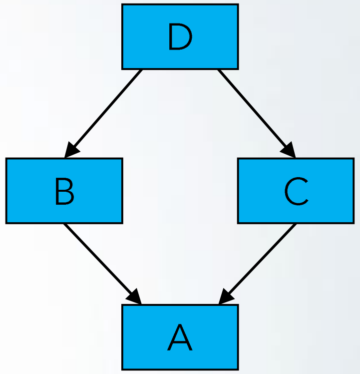
The many addresses of objects

Virtual inheritance (➔ only one part A):

```
struct A {
    virtual ~A() = default;
    int val1;
};

struct B : virtual public A   { int val2; };
struct C : virtual public A   { int val3; };
struct D : public B, public C { int val4; };

D d;
D * pD = &d;    // 0xF868
B * pB = pD;    // 0xF868
C * pC = pD;    // 0xF878 (!)
A * pA_B = pB;
A * pA_C = pC;
```



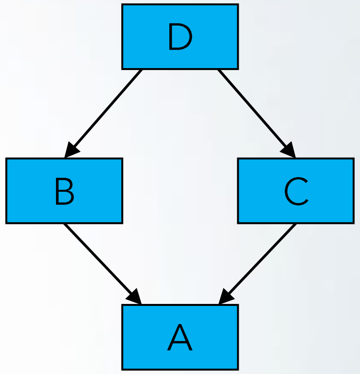
The many addresses of objects

Virtual inheritance (➔ only one part A):

```
struct A {
   virtual ~A() = default;
   int val1;
};

struct B : virtual public A   { int val2; };
struct C : virtual public A   { int val3; };
struct D : public B, public C { int val4; };

D d;
D * pD = &d;    // 0xF868
B * pB = pD;    // 0xF868 (!)
C * pC = pD;    // 0xF878 (!)
A * pA_B = pB;  // 0xF890 (!)
A * pA_C = pC;  // 0xF890
```



➔ Only one A

➔ Different addresses for A, B and C

The many addresses of objects

Question: Is there a way to have different addresses in **single** inheritance?

```
struct A {
    int val1;
};

struct B : public A {
    virtual ~B() = default;
    int val2;
};

B b;
B * pB = &b; // pB = 0xF758
A * pA = pB; // pA = 0xF760
```

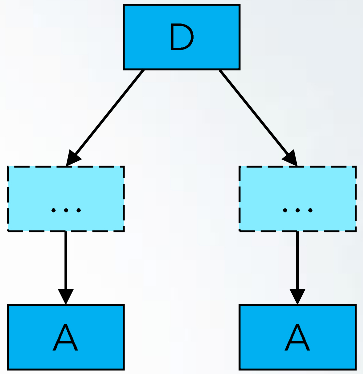
Polymorphic class inherits from non-polymorphic class:

➔ Addresses **not** equal

➔ pA comes **after** pB!
(vtable placed at start)

dynamic_cast<void*>

What is THAT supposed to be?



Question: What is the following doing?

```
PolymorphicType * p = ...;
void * result = dynamic_cast<void*>(p);
```

Answer: `dynamic_cast<void*>` returns a pointer to the **most derived** class pointed to by `p`.
➔ Especially relevant with multiple inheritance

Uses:

- Maintaining a list of „already handled" objects (e.g. used by `boost::serialization`)
- Checking if pointers to base classes point to same object in absence of virtual inheritance

```
bool sameLogicalObject(A * p1, A * p2){
    return dynamic_cast<void*>(p1) == dynamic_cast<void*>(p2);
}
```

„Outer“ catches

„Outer" catches

Question: How to catch an exception from the initializer list **inside** a constructor?

```
struct ThrowingClass {
  ThrowingClass(int someValue) { throw std::exception(); }
};

struct Foo {
  ThrowingClass mObject;
  Foo() : mObject(42) { } // How to catch exception here???
};
```

„Outer" catches

Answer: There is a special construct for this: „Function-try-blocks"

```
struct Foo {
  ThrowingClass mObject;

  Foo()
    try
    : mObject(42)
    { /* Usual body */ }
    catch (std::exception const & ex) {
      // Implicit throw here
    }
};
```

Note: Implicit throw if no explicit throw is given

„Outer“ catches

Unexpected: Works also for normal functions!

```
int Bar()
try {
  // Usual body
}
catch (std::exception const & ex) {
  // Undefined behavior if no return statement
}
```

Uses:

- In constructors: Logging? Exception conversion?
- To confuse colleagues?

Turning classes inside out

Turning classes inside out

Question: Are there legal ways to access `mPrivate` without modifying the class?

```
class Foo
{
private:
  double mPrivate = 3;
};
```

```
cout << f.mPrivate << endl;
```

Turning classes inside out

Idea 1:

```
#define private public
#define class struct
```

➔ Works in practice, but illegal

Turning classes inside out

Idea 2: Exploit same memory layout

```
class Foo {
  double mPrivate = 3;
};

class Mutant {
  public:
    double mTheVariable;
};

int main() {
  Foo f;
  Mutant * m = reinterpret_cast<Mutant*>(&f);
  cout << m->mTheVariable << endl;
}
```

➔ Works in practice, used by `boost::bimap` („mutant idiom“)

➔ But: Not legal according to standard (GotW #76)

Turning classes inside out

Idea 3: Quote from the standard:

„The usual access checking rules do not apply to names used to specify explicit instantiations. "

```
class Foo {
  double mPrivate = 3;
};

using PointerToMember = double Foo::*;

template<PointerToMember offset>
struct Invader { /*... Provide access to offset ... */ };

template struct Invader<&Foo::mPrivate>; // !! Legal !!
```

➔ Legal and generic

➔ Some boilerplate code required in practice (https://stackoverflow.com/a/3173080)

Non-null null pointers

Non-null null pointers

Consider

```
Foo * somePointer = 0;
```

„`0`“ is special here: It means „`nullptr`“, not zero‘ed memory.

➔ Implementation defined, if bit pattern is 0!

➔ Older machines had non-zero pattern (http://c-faq.com/null/machexamp.html).

➔ Nowadays: Member function pointers!

```
struct A;
typedef void (A::*memFuncPtr)();
memFuncPtr ptr = 0;
// Memory of “ptr” is 0 on gcc,
// but non-zero on MSVC ... if „A“ is incomplete
// ... also, sizeof(ptr) > 8 ... depending on „A“
```

Stateful metaprogramming

Does a function evaluated at compile time always return the same?

The value of a constexpr variable is computed at compile-time.

Question: Is the following possible?

```
constexpr int f() { /* ????? */ }

int main() {
  constexpr int a = f();
  constexpr int b = f();

  static_assert(a != b); // Can this every be true?
}
```

I.e. is there stateful metaprogramming?

Expectation: No. There are no re-assignable compile time variables.

Does a function evaluated at compile time always return the same?

```
constexpr int flag(int);

template<class Tag>
struct writer {
  friend constexpr int flag(Tag) {
    return 0;
  }
};

template<bool B, class Tag = int>
struct dependent_writer : writer<Tag> { };

template<
  bool B = noexcept(flag(0)),
  int = sizeof(dependent_writer<B>)
>
constexpr int f() {
  return B;
}
```

```
int main() {
  constexpr int a = f();
  constexpr int b = f();
  static_assert(a != b); // This is true!
}
```

Basic idea:

- `noexcept(...)` (in this case) returns true if a definition of „`...`“ exists
- The definition exists only **after** the instantiation of **`dependent_writer`**, which happens after `noexcept(...)`

➔ `noexcept(...)` gives a different result

Does a function evaluated at compile time always return the same?

More information: http://b.atch.se/posts/non-constant-constant-expressions/

Problems:

- Brittle & abuse of extremly complex template rules
- Reported as defect ➔ Might be fixed in the future

Uses:

- https://github.com/apolukhin/magic_get
  (but also works in ≥C++17 without this „loophole“)
- https://github.com/DaemonSnake/unconstexpr

```
struct somePerson {
  std::string name;
  unsigned birthYear;
};

int main() {
  somePerson val{"Edgar", 1809};
  // Tuple-like access
  unsigned v = boost::pfr::get<1>(val);
  // Output without defining operator<<
  cout << val << endl;
}
```

Thank you for your attention!