An Insight Into Inheritance, Object Oriented Programming, Run-Time Type Information, and Exceptions

PV264 Advanced Programming in C++

Nikola Beneš Jan Mrázek Vladimír Štill

Faculty of Informatics, Masaryk University

Autumn 2020

Motivation

- C++ seems like magic sometimes

Motivation

- C++ seems like magic sometimes
- in fact it is just large
  - C++17 standard is roughly 1600 pages (C++20 ~1850 pages)
  - clang is roughly 600 *k* lines of C++ code and that is just the frontend (it uses LLVM, 800 *k* lines of code, for optimisation and code generation)
  - another 10 *k* of standard library and 8 *k* of runtime library (in case of `libc++`/`libc++abi` from LLVM)

Motivation

- C++ seems like magic sometimes
- in fact it is just large
  - C++17 standard is roughly 1600 pages (C++20 ~1850 pages)
  - clang is roughly 600 *k* lines of C++ code and that is just the frontend (it uses LLVM, 800 *k* lines of code, for optimisation and code generation)
  - another 10 *k* of standard library and 8 *k* of runtime library (in case of `libc++`/`libc++abi` from LLVM)
- and designed for performance
  - one of main principles is that language features should have little to **no performance cost until they are used**
  - this guides design of features such as virtual functions, multiple inheritance, exceptions
- let us now look into some details of the language

Function names in C++

C++ functions can be overloaded, but in the assembly function names have to be unique

Function names in C++

C++ functions can be overloaded, but in the assembly function names have to be unique

- names are mangled, mangled names are unique
- mangling not defined by standard, depends on compiler/platform (gcc/clang/icc use Intel style mangling)
- mangled names contain fully qualified name, argument types
  - `_ZN3foo3barEv` = `foo::bar()`
  - `_ZN3foo3barEi` = `foo::bar(int)`

Function names in C++

C++ functions can be overloaded, but in the assembly function names have to be unique

- names are mangled, mangled names are unique
- mangling not defined by standard, depends on compiler/platform (gcc/clang/icc use Intel style mangling)
- mangled names contain fully qualified name, argument types
  - `_ZN3foo3barEv` = `foo::bar()`
  - `_ZN3foo3barEi` = `foo::bar(int)`
- theoretically, mangled names can be called directly from C
- mangling can be prohibited by using `extern "C"`:

```
namespace foo {
    extern "C" void bar( int ) { /* ... */ }
}
```

  - bar will be callable directly from C, namespace is ignored
  - not recommended to put `extern "C"` functions in namespace
- names can be demangled using `c++filt` (on Linux)

Standard Layout Classes

- “simple” classes that have precisely defined layout
  - can be written to a file and read by a program in another programming language
  - have compatible C counterparts
  - members appear in the class in order of appearance in definition, but there *can be padding* to ensure alignment requirements of some types
    - on `x86_64`, primitive types are usually aligned so that their address is a multiple of their size

Standard Layout Classes

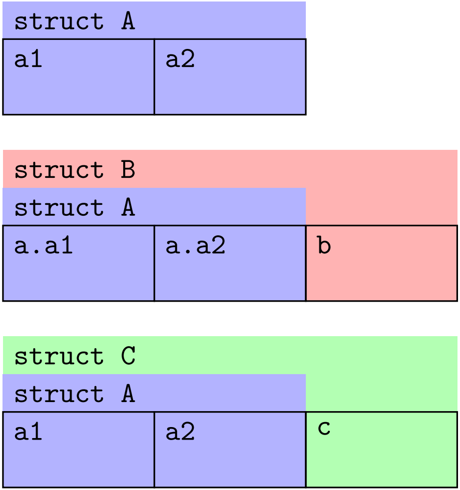
- “simple” classes that have precisely defined layout
  - can be written to a file and read by a program in another programming language
  - have compatible C counterparts
  - members appear in the class in order of appearance in definition, but there *can be padding* to ensure alignment requirements of some types
    - on `x86_64`, primitive types are usually aligned so that their address is a multiple of their size
- generalisation of C++98 Plain Old Data (POD, ∼ C-style structs)
- no virtual functions, virtual base classes, only standard layout data, no mixed access control, ...
- can have (standard layout) base classes
  - non-static data only in one class
- more precisely on cpp reference

In-Memory Layout of Standard Layout Classes

```
struct A {
    int a1;
    int a2;
};

struct B {
    A a;
    int b;
};

struct C : A {
    int c;
};
```



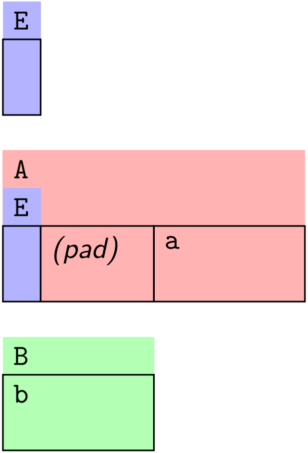
- C is not standard layout
- B and C can have the same in-memory layout (gcc, clang)

Empty Base Class Optimisation

```
struct E { };

struct A {
    E e;
    int a;
};

struct B : E {
    int b;
};
```



- an empty class has size 1
- however, inheriting from an empty class does not increase size
- note: `B` is standard layout

Weird Behaviour of Zero-Sized Arrays

- C allows zero-sized arrays, C++ does not, but GCC & clang support it
  - they are used at the end of variably-sized structures, mainly in POSIX

```
struct A { };
struct B {
    int arr[0];
};

static_assert( sizeof( A ) == 1 );
static_assert( sizeof( B ) == 0 );
```

- zero-sized array has size 0
- putting zero-sized array in an otherwise empty `struct` results in `struct` of size 0

Class Member Functions

in the compiled code, a member function is roughly[1] equivalent to a function that takes an additional first parameter – pointer to `this`

```
struct X {
    int x;
    int foo( int y ) { return x + y; }
};
// code generated by foo is similar to code generated by:
int X_foo( X *this_, int y ) { return this_->x + y; }
```

[1] They can have different calling conventions, but not on x86_64

Class Member Functions

in the compiled code, a member function is roughly[1] equivalent to a function that takes an additional first parameter – pointer to `this`

```
struct X {
    int x;
    int foo( int y ) { return x + y; }
};
// code generated by foo is similar to code generated by:
int X_foo( X *this_, int y ) { return this_->x + y; }
```

- defines function `_ZN1X3fooEi` in assembly, it demangles to `X::foo(int)`
  - mangled the same as `foo( int )` in namespace `X`
- calling is more complex for virtual member functions

[1] They can have different calling conventions, but not on `x86_64`

Class Member Functions

in the compiled code, a member function is roughly[1] equivalent to a function that takes an additional first parameter – pointer to `this`

```
struct X {
    int x;
    int foo( int y ) { return x + y; }
};
// code generated by foo is similar to code generated by:
int X_foo( X *this_, int y ) { return this_->x + y; }
```

- defines function `_ZN1X3fooEi` in assembly, it demangles to `X::foo(int)`
  - mangled the same as `foo( int )` in namespace `X`
- calling is more complex for virtual member functions

but a member function pointer is not the same as a function pointer

[1] They can have different calling conventions, but not on `x86_64`

Class Member Functions

in the compiled code, a member function is roughly[1] equivalent to a function that takes an additional first parameter – pointer to `this`

```
struct X {
    int x;
    int foo( int y ) { return x + y; }
};
// code generated by foo is similar to code generated by:
int X_foo( X *this_, int y ) { return this_->x + y; }
```

- defines function `_ZN1X3fooEi` in assembly, it demangles to `X::foo(int)`
  - mangled the same as `foo( int )` in namespace `X`
- calling is more complex for virtual member functions

but a member function pointer is not the same as a function pointer

- member function pointer must be able to call a virtual function

[1] They can have different calling conventions, but not on `x86_64`

Virtual Member Functions

In object oriented programming it is often necessary to be able to call a function of a derived class through a pointer (or reference) with the type of the base class.

```
struct Base {
  virtual void foo() { std::cout << "Base" << std::endl; }
  virtual ~Base() { }
};
struct Derived : Base {
  void foo() override {
    std::cout << "Derived" << std::endl;
  }
};
int main() {
  std::unique_ptr< Base > b( new Derived() );
  b->foo(); // calls Derived::foo();
}
```

Virtual Functions Implementation

It is not possible to detect at compilation time which version of the virtual function should be called.

- has to be decided at runtime instead
- (single) **dynamic dispatch**

Virtual Functions Implementation

It is not possible to detect at compilation time which version of the virtual function should be called.

- has to be decided at runtime instead
- (single) **dynamic dispatch**
- each class that has any virtual functions contains a *virtual function table* (*vtable*) pointer
  - an additional (usually first) member of the class
  - points to an array of function pointers
  - this array contains pointers to the actual implementations of virtual functions to be used

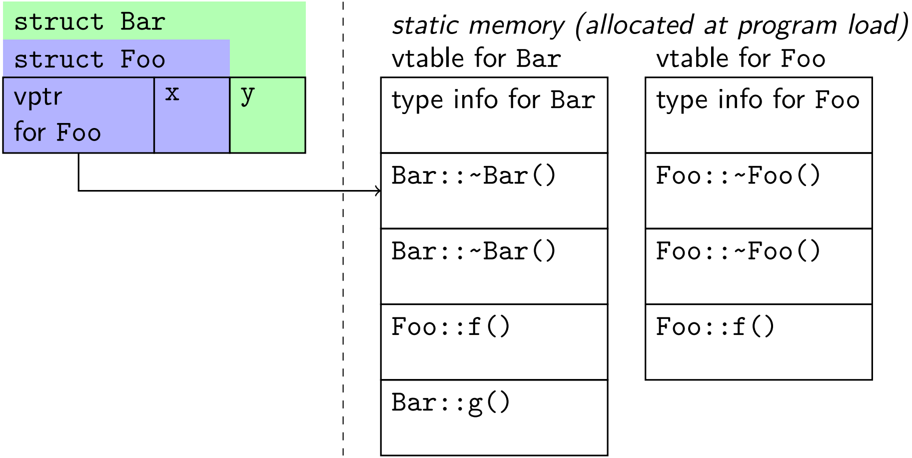
Virtual Functions Implementation

It is not possible to detect at compilation time which version of the virtual function should be called.

- has to be decided at runtime instead
- (single) **dynamic dispatch**
- each class that has any virtual functions contains a *virtual function table* (*vtable*) pointer
  - an additional (usually first) member of the class
  - points to an array of function pointers
  - this array contains pointers to the actual implementations of virtual functions to be used
- see `info vtbl OBJECT` in GDB
- vtable pointer is set in the constructor
- when a member function is called the compiler inserts code that
  1. loads the vtable
  2. finds the appropriate function pointer
  3. calls this function

Virtual Table

```
struct Foo { virtual ~Foo() {}; virtual void f(); int x; };
struct Bar : Foo { void f(); virtual void g(); int y; };
```



- virtual tables are shared by all instances of a given class

Virtual Functions Example

```
struct Base {
  virtual int foo() = 0;
  virtual int bar() = 0;
};
struct Derived : Base {
  int foo() override { return 1; }
  int bar() override { return 2; }
};
void f( Base &x ) { cout << x.bar(); }

// f's implementation is roughly equivalent to (in clang):
void f_lowlevel( Base &x ) {
  using BarPtr = int (*)( Base * );
  BarPtr *vptr = *reinterpret_cast< BarPtr ** >( &x );
  BarPtr barptr = vptr[ BAR_OFFSET ]; // 1 for bar
  cout << barptr( &x );               // 0 for foo
}
```

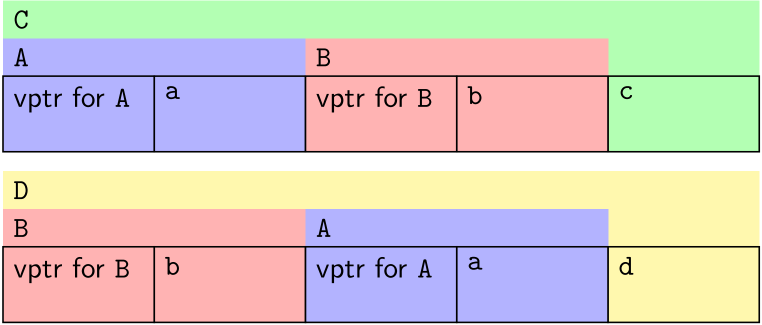
Multiple Inheritance I

- class can have multiple base classes, all of them can define members
- base classes are usually at the beginning of the object, one after another

Multiple Inheritance I

- class can have multiple base classes, all of them can define members
- base classes are usually at the beginning of the object, one after another

```
struct A { long a; virtual void f(); virtual ~A() {} };
struct B { long b; virtual void g(); virtual ~B() {} };
struct C : A, B { long c; void f() override; };
struct D : B, A { long d; void f() override; };
```



Multiple Inheritance II – Casts

```
struct A { long a; virtual void f(); virtual ~A() {} };
struct B { long b; virtual void g(); virtual ~B() {} };
struct C : A, B { long c; void f() override; };
struct D : B, A { long d; void f() override; };

C c; D d;
A &ac = c; A &ad = d; // (1)
C &cac = dynamic_cast< C & >( ac ); // (2)
D &dad = dynamic_cast< D & >( ad );
```

- cast to base class (1) might require adjusting pointer by offset (in case of `ad`)
- cast to derived class should be performed by `dynamic_cast`
  - checks that the object is really a part of the object of target type
  - performs pointer adjustment
  - returns `nullptr` (for pointers) or throws `std::bad_cast` (for references) in case of type failure

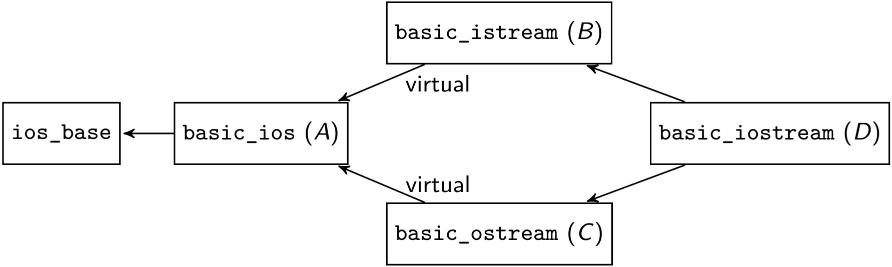
Multiple Inheritance III – Dynamic Dispatch

```
struct A { long a; virtual void f(); virtual ~A() {} };
struct B { long b; virtual void g(); virtual ~B() {} };
struct D : B, A { long d; void f() override; };
D d; d.f();         // (1)
A &ad = d; ad.f(); // (2)
```

`(1)` is a normal dynamic dispatch, but `(2)` is more complicated:

- `ad` points to the `A`-part of `D`
- but `D::f` expects **`this`** to point to `D`
  - cannot be called directly
  - offset could be stored in vtable, but it would need to be checked for any virtual call → slows code even if it does not use multiple inheritance!
  - vtable in `A`-part of `D` contains pointers to wrapper functions that:
    1. adjusts the pointer by constant offset
    2. performs non-virtual call to the actual implementation
  - `B`-part vtable of `D` contains member function pointers directly as it is aligned with `D`

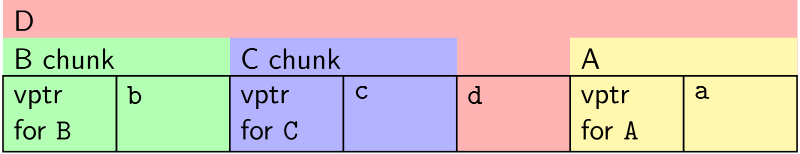
Virtual Inheritance I



- if two base classes (B, C) of class D share common base class (A), then A is duplicated in D
- duplication can be avoided by making B and C inherit from A virtually
- the object hierarchy of such a shape needs to be carefully designed

Virtual Inheritance II

```
struct A { long a; virtual void f(); virtual ~A() {} };
struct B : virtual A { long b; void f() override; };
struct C : virtual A { long c; virtual void g(); };
struct D : B, C { long d; void g() override; };
```



- this is how clang does it, it can differ
- apart from virtual functions, virtual table contains offsets of parts of the struct
  - and again, some virtual functions might be called through wrappers
  - but some wrappers might use dynamic offset

Construction and Destruction Order

Construction order of class with virtual functions

1. construction starts from the base class(es)
   - in order of their appearance, if there are multiple
   - as if the constructor function first called constructors of base classes
2. virtual table pointer(s) are set to point to virtual table(s) of the currently constructed object
3. initializer sections are run
4. constructor body is run

- in case of virtual inheritance, there are also special temporary vtables that are set in base classes while they are being constructed

Construction and Destruction Order

Destruction order of class with virtual functions

1. virtual table pointer(s) are set to point to virtual table(s) of the currently destructed object
2. destructor body is run
3. member data destructors are run
4. base destructors are run (in reverse order of appearance)
   - each of them will reset the appropriate vtable pointers to its vtable

A Note on Destructor Count

- how many destructors does a class have?

A Note on Destructor Count

- how many destructors does a class have? 3

A Note on Destructor Count

- how many destructors does a class have? 3
  1. deleting object destructor – called by `delete foo` expression (D0)
  2. complete object destructor – deletes all data members and all base classes (including virtual) (D1)
  3. base object destructor – deletes all data members and all non-virtual base classes (D2)

A Note on Destructor Count

- how many destructors does a class have? 3
  1. deleting object destructor – called by `delete foo` expression (D0)
  2. complete object destructor – deletes all data members and all base classes (including virtual) (D1)
  3. base object destructor – deletes all data members and all non-virtual base classes (D2)
- destructors D2 and D1 are the same if there are no virtual base classes
- destructor D0 first destroys the object (using D1) then calls `operator delete` to free the memory
  - `operator delete` can be overloaded in a class, so this ensures the right one is called
  - (`operator new` can also be overloaded in a class)
- note: not C++ standard, this is Intel ABI (`clang` on Linux, `gcc`)

What About Constructors?

- similarly, class has a complete object constructor (C1), a base object constructor (C2) that is called from the descendant's constructor, and an allocating object constructor (C3)
  - again C1 and C2 are the same unless virtual inheritance takes place
- allocating constructor/destructor might be missing

Member Function/Data Pointers

- normal data and function pointers are essentially the same thing – address of memory where data or code is stored (on modern architectures)
- but it can be useful to have "pointers" into a class, or to a function of a class – how?

Member Function/Data Pointers

- normal data and function pointers are essentially the same thing – address of memory where data or code is stored (on modern architectures)
- but it can be useful to have "pointers" into a class, or to a function of a class – how?
  - `int Foo::*a` – a pointer to data member of type `int` that belongs to class `Foo` and is called `a`
  - `void (Foo::*f)( int )` – a pointer to member function of `Foo` that returns `void` and gets `int`; the pointer is called `f`

Member Function/Data Pointers

- normal data and function pointers are essentially the same thing – address of memory where data or code is stored (on modern architectures)
- but it can be useful to have “pointers” into a class, or to a function of a class – how?
  - `int Foo::*a` – a pointer to data member of type `int` that belongs to class `Foo` and is called `a`
  - `void (Foo::*f)( int )` – a pointer to member function of `Foo` that returns `void` and gets `int`; the pointer is called `f`
- may not really be an address – implementation can differ
  - for non-virtual functions usually contains address directly
  - for pointer to virtual member function it is necessary to do vtable lookup by function index
  - in case of multiple inheritance, offset to the right vtable is also needed

Member Function/Data Pointers – example

```
struct Foo { int bar(); int baz() const; };
int main() {
    int (Foo::*pa)() = &Foo::bar; // & is necessary
    // pointer to const member function
    int (Foo::*pb)() const = &Foo::baz;

    Foo f;
    Foo *fptr = &f;

    int x = (f.*pb)(); // using member function pointer
    int y = (fptr->*pa)(); // the same on pointer
}
```

Exceptions

- exceptions are important for handling errors (such as resource allocation failures) in a clean way
  - repeated manual error checking can be avoided
  - normal control-flow of functions is not cluttered

Exceptions

- exceptions are important for handling errors (such as resource allocation failures) in a clean way
  - repeated manual error checking can be avoided
  - normal control-flow of functions is not cluttered
- but they also come at cost
  - to readability – error handling code can be far from error producing code
  - to speed – they are slower if an error occurs
- still, code with exceptions used for rare errors will be probably better readable

Exceptions

- exceptions are important for handling errors (such as resource allocation failures) in a clean way
  - repeated manual error checking can be avoided
  - normal control-flow of functions is not cluttered
- but they also come at cost
  - to readability – error handling code can be far from error producing code
  - to speed – they are slower if an error occurs
- still, code with exceptions used for rare errors will be probably better readable
- there are many possibilities to implement exceptions
  - checkpointing – CPU registers are saved before a function that can throw is executed, restored if exception is raised (old)
  - **“zero-cost exceptions”** – should have no performance overhead compared to code without error checking

Zero-Cost Exceptions

- under normal circumstances no exception-related code is executed
- handled by C++ runtime library
  - implementation can differ, clang/`libc++abi` implementation for `x86_64` Linux is described here
  - `libsupc++` from GCC works similarly

Zero-Cost Exceptions

- under normal circumstances no exception-related code is executed
- handled by C++ runtime library
  - implementation can differ, clang/`libc++abi` implementation for `x86_64` Linux is described here
  - `libsupc++` from GCC works similarly
- when `throw` is executed:
  1. an exception object is allocated (on heap, or in emergency storage = global variable)
  2. the unwinder library is invoked to handle stack search and actual transfer of control (*unwinding*)

Unwinding Basics I

- unwinder is provided by the platform, it is not C++ specific
- extensively uses metadata tables generated by the compiler to find
  - boundaries of stack frames
  - which function corresponds to which stack frame
  - how to search for handlers in given function and frame

Unwinding Basics I

- unwinder is provided by the platform, it is not C++ specific
- extensively uses metadata tables generated by the compiler to find
  - boundaries of stack frames
  - which function corresponds to which stack frame
  - how to search for handlers in given function and frame
- cooperates with language's runtime library to find handler
  - language defines *personality routine* that is called by the unwinder to find handlers
  - personality uses metadata tables for given function (found by unwinder) to find the right handler

Unwinding Basics II

- two kinds of exception handlers
  - catch handlers – end exception propagation, resolve exception
    - `catch`
    - exception specification
  - cleanup handlers – perform cleanup, exception propagation continues afterwards
    - call destructors
    - triggered only if a catch handler is found[2]

[2] not specified by standard but common on Linux/Unix

Unwinding Basics II

- two kinds of exception handlers
  - catch handlers – end exception propagation, resolve exception
    - `catch`
    - exception specification
  - cleanup handlers – perform cleanup, exception propagation continues afterwards
    - call destructors
    - triggered only if a catch handler is found[2]
- which catch handler is appropriate is detected from run-time type information (RTTI) that encodes the inheritance hierarchy
- cost comes from
  - cost of actual unwinding and related metadata search and decoding
  - cost of inspecting the type hierarchy of the exception

[2] not specified by standard but common on Linux/Unix

Exception Specification (`throw(...)`, `noexcept`)

```
int foo() throw ( std::bad_alloc );
int bar() noexcept;
```

- `throw()`, `throw(exception types, ...)`
  - specifies that function is allowed to throw only specified types
  - throwing any other type results in termination of program
  - deprecated in C++11
  - second version removed in C++17, first made equivalent to `noexcept`

Exception Specification (`throw(...)`, `noexcept`)

```
int bar() noexcept;
template< typename T > int baz() noexcept(
            std::is_nothrow_constructible< T >::value );
```

- `noexcept`
  - specifies the function is not allowed to throw
  - not checked by the compiler, but throwing from `noexcept` function will terminate the program (using `std::terminate`)
  - compiler-generated default constructors, move and copy constructors are `noexcept` by default
    - unless appropriate base class or member constructors are not
  - destructors are `noexcept` unless *explicitly* marked otherwise
- `noexcept( EXPR )`
  - specifies function is not allowed to throw if `EXPR` evaluates to `true`
  - `noexcept` is equivalent to `noexcept(true)`

Implications of `noexcept`

- certain operations can be safely performed only if a function is `noexcept`
  - vector can use move construction when growing only if move constructor is `noexcept`
    - exception in move constructor would leave vector in inconsistent state
  - the presence of `noexcept` can impact performance
- move constructors should be `noexcept` if possible

Uncaught Exception Handler

- if an exception is not caught `std::terminate` is called
- `std::terminate` defaults to killing the program, but can be customised
  - `std::set_terminate`
  - useful for logging exceptions
  - should not try to restore execution (`catch` is for that)

Run-Time Type Information I

- underlying mechanism for the implementation of `dynamic_cast` and exception matching in `catch` clauses

Run-Time Type Information I

- underlying mechanism for the implementation of `dynamic_cast` and exception matching in `catch` clauses

```
#include <typeinfo> // necessary for use of typeid
...
auto &tint = typeid( int ); // (1)
auto &texpr = typeid( 1 + 1 ); // (2)
Foo x; // Foo has virtual functions
auto &tfoo = typeid( x ); // (3)
```

- `typeid(arg)` returns constant reference to `std::type_info` object representing type of its argument
  - if `arg` is a type, returned `type_info` describes this type `(1)`
  - if `arg` is an expression of apolymorphic type, `type_info` of runtime type of this exception is returned `(3)`
    - polymorphic type = has virtual method(s)
  - otherwise `type_info` for static type of the expression is returned `(2)`

Run-Time Type Information II

- `std::type_info`
  - defines the `name` method that is used to get the (implementation defined) name of the type
    - on Linux a part of the mangled name
  - operators ==, `!=` for checking if the corresponding types are equal
  - not constructible, copyable
  - stored in static memory (generated by compiler)
  - pointer to `type_info` is present in virtual function table of polymorphic objects
- `std::type_index`
  - hashable and comparable wrapper around `type_info` that can be used as a key for associative maps (`std::map`, `std::unordered_map`)

Multiple Dispatch (Multimethods)

- `virtual` functions provide *single dispatch*
  - the function to be called depends on the dynamic type of *one* parameter (the current object, `*this`)
- **multiple dispatch:** the function depends on the dynamic type of *multiple* parameters; why is it useful?

Multiple Dispatch (Multimethods)

- `virtual` functions provide *single dispatch*
  - the function to be called depends on the dynamic type of *one* parameter (the current object, `*this`)
- **multiple dispatch:** the function depends on the dynamic type of *multiple* parameters; why is it useful?
  - interaction between various kinds of objects
    (imagine a computer game with weapons and monsters or a vector graphics library that computes intersection of shapes)

Multiple Dispatch (Multimethods)

- `virtual` functions provide *single dispatch*
  - the function to be called depends on the dynamic type of *one* parameter (the current object, `*this`)
- **multiple dispatch:** the function depends on the dynamic type of *multiple* parameters; why is it useful?
  - interaction between various kinds of objects
    (imagine a computer game with weapons and monsters or a vector graphics library that computes intersection of shapes)
- some languages support multiple dispatch natively
  - Common Lisp, Perl 6, C# 4.0, ...
- other languages have library support for multiple dispatch
  - C, C++, Java, Perl, Python, ...
- how to emulate multiple dispatch in C++?

Multiple Dispatch (Multimethods)

- `virtual` functions provide *single dispatch*
  - the function to be called depends on the dynamic type of *one* parameter (the current object, `*this`)
- **multiple dispatch:** the function depends on the dynamic type of *multiple* parameters; why is it useful?
  - interaction between various kinds of objects (imagine a computer game with weapons and monsters or a vector graphics library that computes intersection of shapes)
- some languages support multiple dispatch natively
  - Common Lisp, Perl 6, C# 4.0, ...
- other languages have library support for multiple dispatch
  - C, C++, Java, Perl, Python, ...
- how to emulate multiple dispatch in C++?
  - `dynamic_cast`
  - multidimensional "virtual tables"
  - for *double dispatch*: **visitor pattern**

Visitor Pattern

- base element class `Element`
  - one purely virtual method `accept(Visitor&)`
- base visitor class `Visitor`
  - a virtual method `visit(ConcreteElement&)` for each concrete child of `Element`

- children of `Element` override `accept` as follows:

```
struct Dragon : Element {
    void accept(Visitor& v) override { v.visit(*this); }
};
```

- children of `Visitor` may override its virtual methods

```
struct Axe : Visitor {
    void visit(Dragon&) override { /* ... */ }
    void visit(Troll&) override { /* ... */ }
    // ...
};
```