Virtual functions & visitors

Virtual functions

- The set of virtual functions (and their signatures) is fixed by the definition of the base class
  - Virtual functions can not be templates
- The required functionality must be anticipated when the base class is defined
  - Any extensions change the interface between the base class and any derived classes – any change is expensive
- All the implementations of a virtual function are present in the executable code, even if the virtual function is never called
  - The code generation for a virtual function is triggered by the generation of a constructor of the enclosing class
- Acceptable when functionality is fixed and the set of different implementations (concrete classes) grows
  - E.g. printer drivers
- Problematic when the set of concrete classes is fixed and the required functionality grows
  - E.g. intermediate code (AST) objects inside an optimizing compiler

- Forward-declare all concrete object classes

```
class ConcreteObject1; class ConcreteObject2;
```

- Define the abstract visitor class with a virtual function for every concrete object class
  - Traditionally named visit, sometimes distinguished as visitConcreteObject1, ...

```
class AbstractVisitor { public:
  virtual ~AbstractVisitor() noexcept {}
  virtual void visit(ConcreteObject1 & x) = 0;
  virtual void visit(ConcreteObject2 & x) = 0;
};
```

- Define a virtual function in the abstract object class, accepting a reference to the abstract visitor
  - Traditionally named accept
  - Other virtual functions may be declared in the abstract class for direct invocation without a visitor

```
class AbstractObject { public:
  virtual ~AbstractObject() noexcept {}
  virtual void accept(AbstractVisitor & v) = 0;
  /*...*/
};
```

- Implement the accept function in every concrete object class, calling the corresponding visit function

```
class ConcreteObject1 : public AbstractObject {
  virtual void accept(AbstractVisitor & v) override { v.visit(*this); }
  /*...*/
};
class ConcreteObject2 : public AbstractObject {
  virtual void accept(AbstractVisitor & v) override { v.visit(*this); }
  /*...*/
};
```

Visitor pattern

```
class AbstractVisitor { public:
  virtual void visit(ConcreteObject1 & x) = 0;
  virtual void visit(ConcreteObject2 & x) = 0;
};
class AbstractObject { public:
  virtual void accept(AbstractVisitor & v) = 0;
};
class ConcreteObject1 : public AbstractObject {
  virtual void accept(AbstractVisitor & v) override { v.visit(*this); }
};
/*...*/
```

- Usage: define a concrete visitor class for every action required
  - Implement the visit function for every concrete object class
  - Data elements may be present in the concrete visitor (cf. capture in functors)

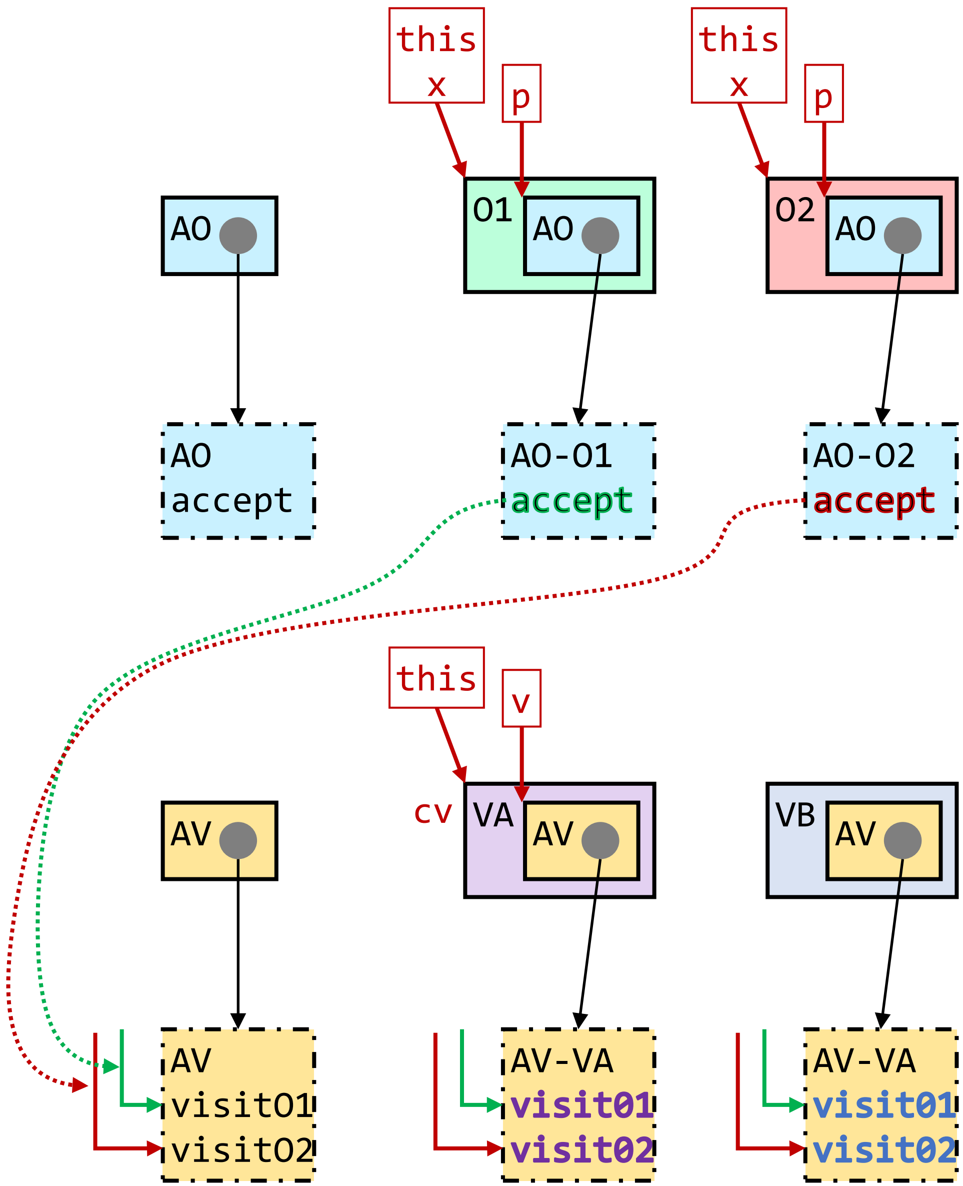
```
class ConcreteVisitorA : public AbstractVisitor {
  virtual void visit(ConcreteObject1 & x) override { /*...*/ }
  virtual void visit(ConcreteObject2 & x) override { /*...*/ }
  /*...*/
};
```

  - Instantiate the concrete visitor and pass it to the accept function
    - Fill visitor data elements before and/or retrieve their values after calling accept

```
void invokeA(AbstractObject * p) {
  ConcreteVisitorA cv;
  /* fill cv... */
  p->accept(cv);
  /* retrieve from cv... */
}
```

Visitor pattern - memory layout

```
class AV { public:
  virtual void visit(O1 & x) = 0;
  virtual void visit(O2 & x) = 0;
};
class AO { public:
  virtual void accept(AV & v) = 0;
};
class O1 : public AO {
  virtual void accept(AV & v)
    override { v.visit(*this); }
};
class O2 : public AO {
  virtual void accept(AV & v)
    override { v.visit(*this); }
};
class VA : public AV {
  virtual void visit(O1 & x)
    override { A_1(x); }
  virtual void visit(O2 & x)
    override { A_2(x); }
};
class VB : public AV {
  virtual void visit(O1 & x)
    override { B_1(x); }
  virtual void visit(O2 & x)
    override { B_2(x); }
};
AO * p = /*...*/;
VA cv;
p->accept(cv);
```



Visitor pattern functionality

- Instantiate the concrete visitor and pass it to the accept function

```
ConcreteVisitorA cv;
```

- The concrete visitor object is initialized with type information according to the action
  - Plus any data required for the action
  - This is one of the few legitimate cases of non-dynamically allocated object with inheritance

```
p->accept(cv);
```

- Implicit cast of the visitor reference from the concrete visitor to the abstract visitor
  - The code “forgets” the action required (it remains remembered in the run-time data)
- Virtual call to accept, using the type information stored in the abstract object (*p)
  - Dispatch to the implementation corresponding to the object type
  - Simultaneously, cast this pointer from the abstract object (*p) to a concrete object

```
void ConcreteObject1::accept(AbstractVisitor & v) {
  v.visit(*this);
```

- Virtual call to visit, using the type information stored in the abstract visitor (v)
  - Dispatch to the implementation corresponding to the visitor type
  - Simultaneously, cast this pointer from the abstract visitor (v) to the concrete object

```
void ConcreteVisitorA::visit(ConcreteObject1 & x) {
```

- The code being executed is now specialized to both the object type and the action required
  - A case of *double-dispatch*

Set of virtual functions

- Fixed set of actions
  - Interface of the abstract object
- Extensible set of object types
  - New concrete object classes may be defined without modifying the abstract class

Visitor pattern

- Extensible set of actions
  - New concrete visitors may be defined without modifying abstract (object and visitor) classes
- Fixed set of object types
  - Interface of the visitor
- Higher run-time cost
  - Two indirect calls

Visitor and lambda

Visitor pattern and lambda

- A concrete visitor may invoke a functor/lambda

```
template< typename F>
class FunctorVisitor : public AbstractVisitor {
public:
  FunctorVisitor( F f) : f_( std::move(f)) {}
private:
  F f_;
  virtual void visit(ConcreteObject1 & x) override { f_(x); }
  virtual void visit(ConcreteObject2 & x) override { f_(x); }
};
```

- Statically polymorphic functor/auto lambda is required
  - accept and visit are virtual – *run-time polymorphism*
    - accept dispatches according to the object type
    - visit dispatches according the functor/lambda type (forgotten through accept)
  - operator() is a template (due to the auto argument) – *compile-time polymorphism*
    - compiler creates two implementations of operator() called by f_(x)

```
void invokeA(AbstractObject * p) {
  FunctorVisitor cv([](auto && x){ x.something(); });
  p->accept(cv);
}
```

  - *Class Template Argument Deduction* [C++17]: compiler derives the class template argument F from the type of constructor argument in the initialization
  - both concrete objects must contain (non-virtual) member function something

Visitor pattern and lambda

- operator() may be overloaded in a functor – *compile-time polymorphism*

```
struct ftorA {
  void operator()(ConcreteObject1 & x) { x.something(); }
  void operator()(ConcreteObject2 & x) { x.something_else(); }
};
void invokeA(AbstractObject * p) {
  FunctorVisitor cv(ftorA());
  p->accept(cv);
}
```

- polymorphic ftorA vs. concrete visitor:
  - ftorA is not derived from AbstractVisitor, operator() are not virtual
  - non-virtuality allows tricks impossible with visitors:

```
struct ftorA {
  void operator()(ConcreteObject1 & x) { x.something(); }
  template<typename O> void operator()(O && x) { x.something_default(); }
};
```

  - especially useful if the object-type hierarchy contains intermediate abstract objects:

```
struct ftorA {
  void operator()(IntermediateObjectA & x) { x.something_in_A(); }
  void operator()(IntermediateObjectB & x) { x.something_in_B(); }
  template<typename O> void operator()(O && x) { x.something_default(); }
};
```

- the functor may contain less functions than an equivalent visitor
  - simplest cases may be handled by a lambda
- run-time cost is roughly the same as for visitors
  - dominated by the two virtual calls accept+visit, the non-virtual calls are negligible

Visitor pattern

- The statically-polymorphic interface may be wrapped into the AbstractObject
  - It makes the underlying visitor mechanism invisible

```
class AbstractObject {
public:
  virtual ~AbstractObject() noexcept {}

  template<typename F> void accept_static(F f)
  { accept( FunctorVisitor(std::move(f))); }

private:
  virtual void accept(const AbstractVisitor & v) = 0;
};
```

  - Usage:

```
p->accept_static([](auto && x){ x.something(); });
```

- The statically-polymorphic functors (supplied to accept_static) may sometimes adapt automatically to a new concrete object type
  - The underlying AbstractVisitor and FunctorVisitor must still be manually adjusted

- The (polymorphic) functor is usually passed by value
  - It does not support self-modifying functors (mutable lambdas)
  - Programmers are used to this from library algorithms etc.
- Visitors must always passed by reference
  - Otherwise the virtual functions would not work
  - This corresponds to the observation that objects with inheritance have their identity

- Trick: Combining lambdas into a polymorphic functor

```
template< typename F1, typename F2>
class mixer : public F1, public F2 {
public:
  mixer(F1 f1, F2 f2) : F1(std::move(f1)), F2(std::move(f1)) {}
  using F1::operator();
  using F2::operator();
};
```

  - The using clauses hoist the operators into a common scope
    - Otherwise, calling operator() will fail due to ambiguity
    - overload resolution is done only after determining single class scope for the called function

```
template< typename F1, typename F2> auto operator|(F1 f1, F2 f2)
{ return mixer<F1,F2>(std::move(f1),std::move(f2)); }
```

  - Usage:

```
p->accept_static(
   [](ConcreteObject1 & x){ x.something(); }
  |[](auto && x){ x.something_default(); }
);
```

  - **Beware**: This is **grossly ineffective** for functors containing (the same) data

```
p->accept_static(
   [a,b,c](ConreteObject1 & x){ x.something(a,b,c); }
  |[a,b,c](auto && x){ x.something_default(a,b,c); }
);
```

std::variant

- **std::variant** - a polymorphic type containing one of a fixed set of types
    - Safe replacement of C unions
  - plus a data element to remember which type it is

```
using VT = std::variant< T0, T1, T2>;
```

  - More space-effective than a pointer to a dynamically-allocated object with inheritance
    - If the types are of similar sizes
    - But not extensible
  - There is no common interface (base class) required in the types
  - Allows assignment from any of the types

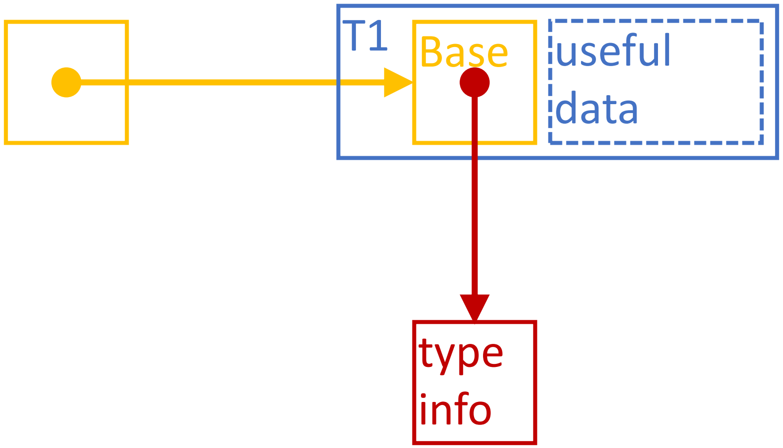
```
T0 v0 = /*...*/;
VT a = v0;
VT b(std::in_place_type<T1>, /*...*/);   // calls T1::T1(/*...*/)
VT c(std::in_place_index<2>, /*...*/);   // calls T2::T2(/*...*/)
b = v0;                  // calls T1::~T1(), T0::T0(v0)
c.emplace<T1>(/*...*/); // calls T2::~T2(), T1::T1(/*...*/)
a.emplace<2>(/*...*/);  // calls T0::~T0(), T2::T2(/*...*/)
```

  - Access to the contained data:

```
void action( VT & vo)
{
  switch ( vo.index() ) {
  case 0: { T0 & v0 = std::get< 0>(vo); v0.f(); } break;
  case 1: { T1 & v1 = std::get< 1>(vo); v1.g(); } break;
  case 2: { T2 & v2 = std::get< 2>(vo); v2.h(); } break;
}
```

Storing polymorphic objects

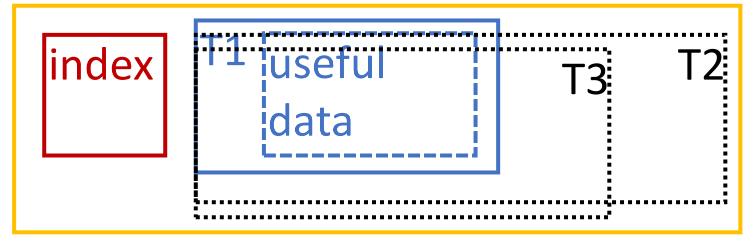
Using inheritance




```
std::unique_ptr<Base> x =
std::make_unique<T1>(/*...*/);
```

- Higher run-time cost
  - Pointer, dynamic allocation
- Extensible set of concrete types

Using std::variant




```
std::variant<T1,T2,T3> x =
T1(/*...*/);
```

- Lower run-time cost
  - No dynamic allocation
- Always requires maximum space
- Not intrusive
  - Does not use any base class
- Fixed set of alternative types

std::variant and static visitor

```
using VT = std::variant< T0, T1, T2>;
```

- std::visit - Usage through a polymorphic functor

```
struct VisitorA {
  void operator()( T0 & x) { /*...*/ }
  void operator()( T1 & x) { /*...*/ }
  void operator()( T2 & x) { /*...*/ }
};
void action( VT & vo)
{
  VisitorA va;
  std::visit(va, vo);
}
```

- It may be used with a polymorphic lambda

```
std::visit([](auto && a){ a.something(); }, vo);
```

  - All the types need a common interface
    - The interface is not explicitly declared
  - This is a static equivalent of inheritance and virtual functions
    - At higher cost – visit requires a tricky implementation similar to visitors
    - But the memory footprint may still be smaller – no dynamic allocation