Compiling, linking issues

- Inclusion nightmare and compile time
- How to reduce included files
- PIMPL
- Fast PIMPL
- Template code blow
- Linking C and C++ together
- Order of linking issues
- Static initialization/destruction problem

Header files

- Good for defining interface
- Good for breaking circular dependencies
- Must for templates
- Increase compile time
- Break OO principles

Header files

```
#include <iostream>
#include <ostream>
#include <list>
// none of A, B, C, D, E are templates
// Only A and C have virtual functions
#include "a.h"      // class A
#include "b.h"      // class B
#include "c.h"      // class C
#include "d.h"      // class D
#include "e.h"      // class E

class X : public A, private B
{
public:
        X( const C&);
    B   f(int, char*);
    C   f(int, C);
    C&  g(B);
    E   h(E);
    virtual std::ostream& print(std::ostream&) const;
private:
    std::list<C>    clist_;
    D               d_;
};
```

Header files

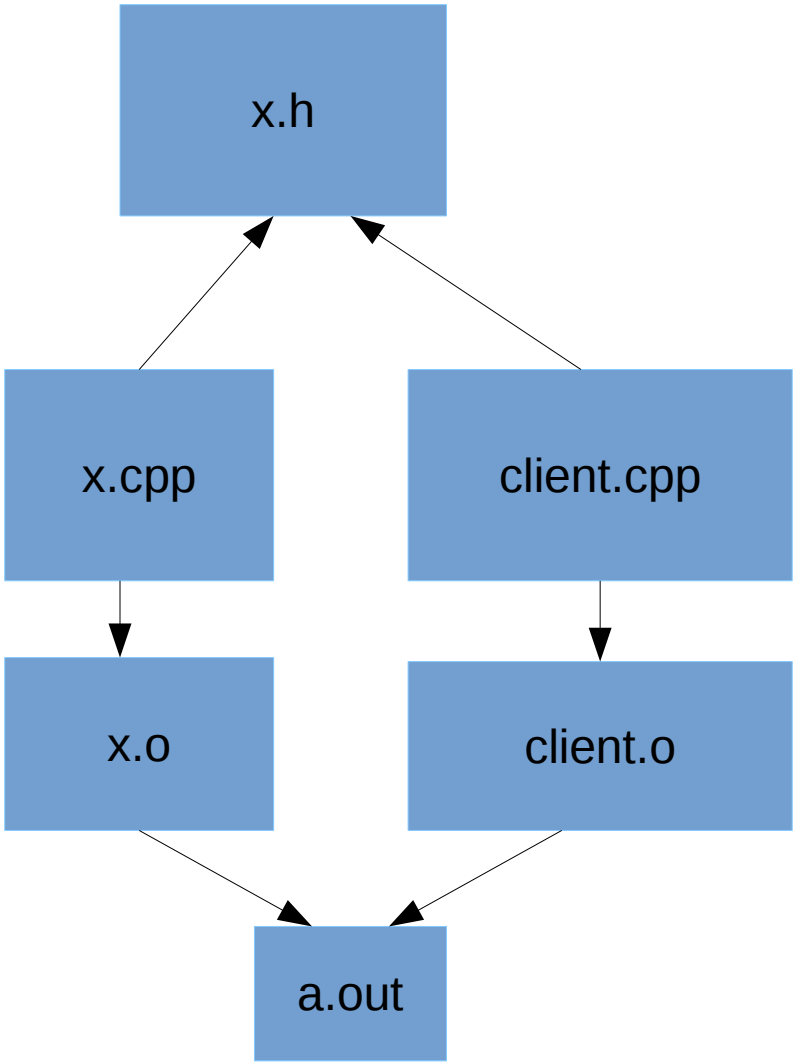
- Remove <iostream> People automatically include <iostream>, even if input functions never used.
- Replace <ostream> with <iosfwd>. Parameters and return types only need to be forward declared. Because ostream is basic_ostream<char> template, it is not enough to declare.
- Replace "e.h" with forward declaration of class E.
- Leave "a.h" and "b.h": we need a full declaration of the base classes in case of inheritance. The compiler must know the size of bases, whether functions are virtual or not.
- Leave "c.h" and "d.h": list<C> and D are private data members of X.

Header files v2

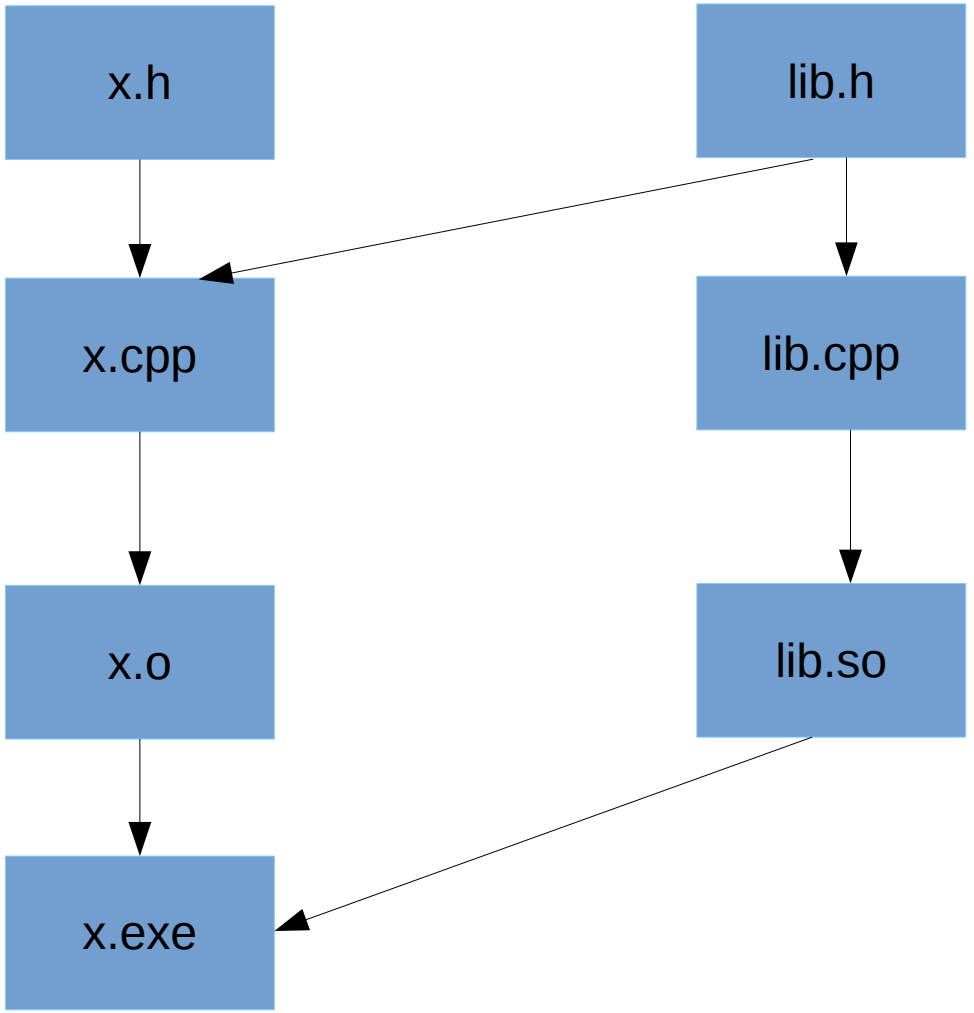
```
#include <iosfwd>
#include <list>
// none of A, B, C, D, E are templates
// Only A and C have virtual functions
#include "a.h"      // class A
#include "b.h"      // class B
#include "c.h"      // class C
#include "d.h"      // class D

class E;
class X : public A, private B
{
public:
        X( const C&);
    B   f(int, char*);
    C   f(int, C);
    C&  g(B);
    E   h(E);
    virtual std::ostream& print(std::ostream&) const;
private:
    std::list<C>    clist_;
    D               d_;
};
```

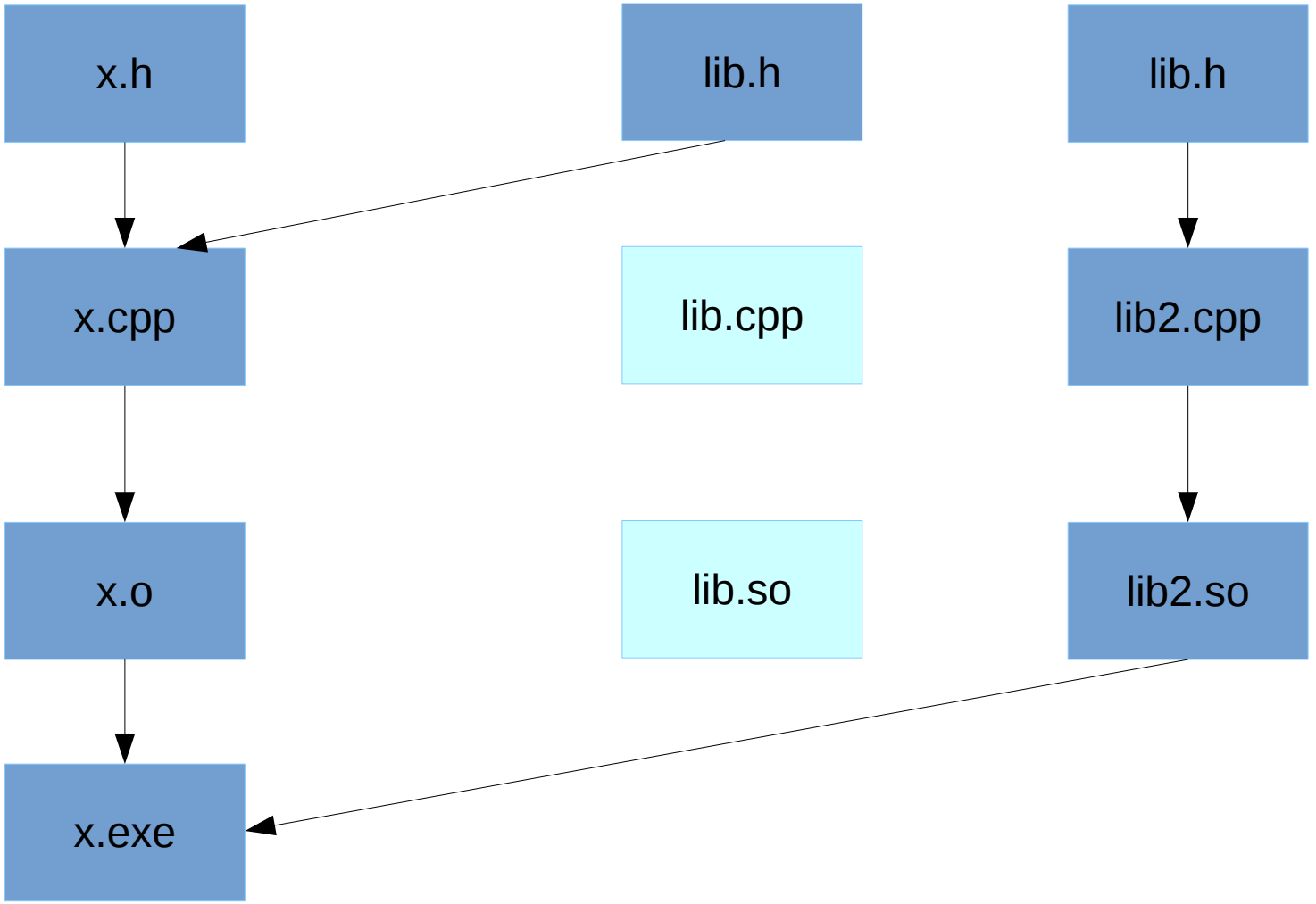
PIMPL



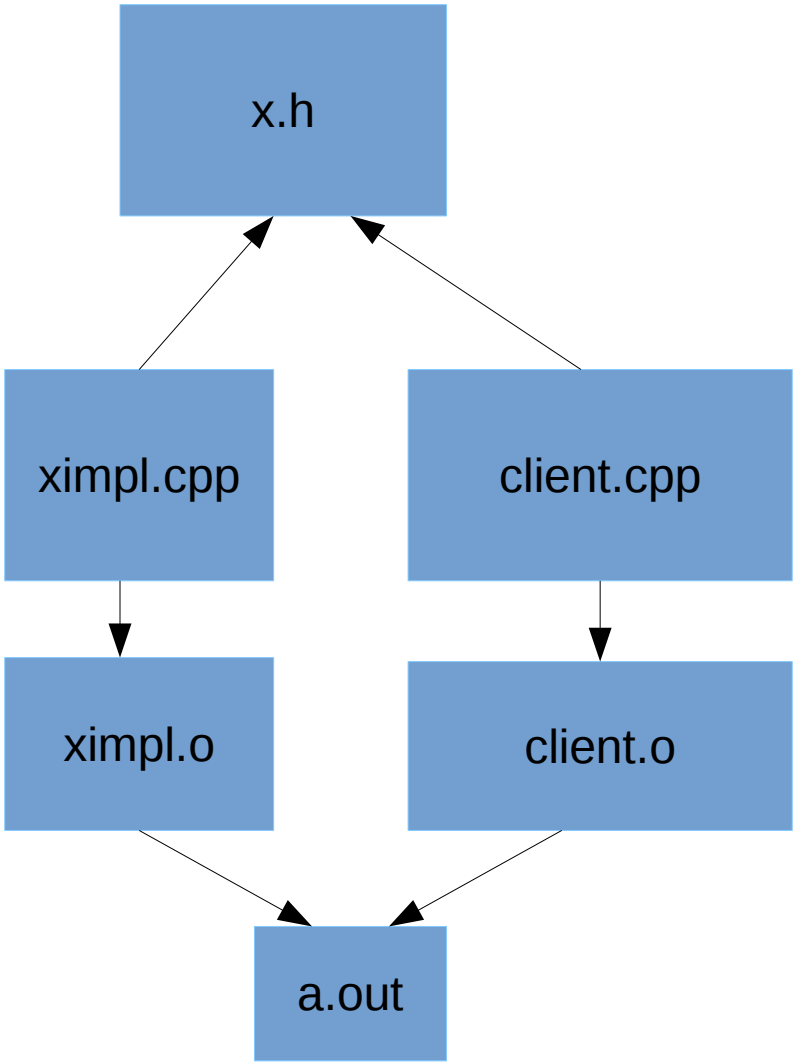
Binary compatibility



Binary compatibility



PIMPL



PIMPL

```
// file x.h
class X
{
    // public and protected members
private:
    // pointer to forward declared class
    struct Ximpl;
    XImpl *pimpl_;  // opaque pointer
};

// file ximpl.cpp
struct XImpl   // not neccessary to declare as "class"
{
    // private members; fully hidden
    // can be changed at without
    // recompiling clients
};
```

PIMPL

```
#include <iosfwd>
#include "a.h"      // class A
#include "b.h"      // class B
class C;
class E;
class X : public A, private B
{
public:
        X( const C&);
    B   f(int, char*);
    C   f(int, C);
    C&  g(B);
    E   h(E);
    virtual std::ostream& print(std::ostream&) const;
private:
    struct Ximpl;
    XImpl *pimpl_; // opaque pointer to forward-declared class
};
// file x.cpp
#include "x.h"
#include "c.h"      // class C
#include "d.h"      // class D
struct Ximpl
{
    std::list<C>    clist_;
    D               d_;
};
```

PIMPL with unique_ptr

```
// file x.h
class X
{
public:
    X();
private:
    // pointer to forward declared class
    struct Ximpl;
    unique_ptr<Ximpl> pimpl_;  // opaque pointer
};

// file ximpl.cpp
struct X::XImpl   // not neccessary to declare as "class"
{
    X::X()
    // can be changed at without
    // recompiling clients
};
X::X() : pimpl_(std::make_unique<XImpl>()) {}
```

PIMPL with unique_ptr

```
#include “x.h”

X  xObj;   // Error: incomplete type!
```

PIMPL with unique_ptr

```
// file x.h
class X
{
public:
    X();
    ~X();  // declaration only!
private:
    // pointer to forward declared class
    struct Ximpl;
    unique_ptr<Ximpl> pimpl_;  // opaque pointer
};

// file ximpl.cpp
struct X::XImpl   // not neccessary to declare as "class"
{
    X::X()
    // can be changed at without
    // recompiling clients
};
X::X() : pimpl_(std::make_unique<XImpl>()) {}
X::~X() {}  // definition, also works: X::~X() = default
```

PIMPL with unique_ptr

```
// file x.h
class X
{
public:
    X();
    ~X();
    X(X&& rhs);
    X& operator=(X&& rhs);
private:
    // pointer to forward declared class
    struct Ximpl;
    unique_ptr<Ximpl> pimpl_;  // opaque pointer
};

// file ximpl.cpp
X::X() : pimpl_(std::make_unique<XImpl>()) {}
X::~X() = default;
X::X(X&& rhs) = default;
X& X::operator=(X&& rhs) = default;
```

PIMPL with unique_ptr

```
// file x.h
class X
{
public:
    X();
    ~X();
    X(X&& rhs);
    X& operator=(X&& rhs);
    X(const X& rhs);
    X& operator=(const X& rhs);
private:
    // pointer to forward declared class
    struct Ximpl;
    unique_ptr<Ximpl> pimpl_;  // opaque pointer
};

// file ximpl.cpp
X::X(const X& rhs) : pimpl_(std::make_unique<XImpl>(*rhs.pImpl_) {}
X& X::operator=(const X& rhs){ *pImpl_ = *rhs.pImpl_; return *this; }
```

Removing inheritance

```
#include <iosfwd>
#include "a.h"      // class A
class B;
class C;
class E;
class X : public A  // ,private B
{
public:
        X( const C&);
    B   f(int, char*);
    C   f(int, C);
    C&  g(B);
    E   h(E);
    virtual std::ostream& print(std::ostream&) const;
private:
    struct Ximpl;
    XImpl *pimpl_; // opaque pointer to forward-declared class
};
// file x.cpp
#include "x.h"
#include "b.h"      // class B
#include "c.h"      // class C
#include "d.h"      // class D
struct Ximpl
{
    B               b_;
    std::list<C>    clist_;
    D               d_;
};
```

Fast PIMPL

```
// file x.h
class X
{
  // public and protected members
private:
  static const size_t XImplSize = 128;
  char ximpl_[XImplSize]; // instead opaque pointer
};

// file ximpl.cpp
struct XImpl   // not neccessary to declare as "class"
{
  Ximpl::Ximpl(X *tp) : _self(tp) {
    static_assert (XImplSize >= sizeof(XImpl));
    // ...
  };
  X *_self; // might be different than XImpl::this
};
X::X()  { new (ximpl_) XImpl(this); }
X::~X() { (reinterpret_cast<XImpl*>(ximpl_)->~XImpl(); }
```

Fast PIMPL

```
// file x.h
class X
{
  // public and protected members
private:
  static const size_t XImplSize = 128;
  alignas(std::max_align_t) char ximpl_[XImplSize];
};

// file ximpl.cpp
struct XImpl   // not neccessary to declare as "class"
{
  Ximpl::Ximpl(X *tp) : _self(tp) {
    static_assert (XImplSize >= sizeof(XImpl));
    // ...
  };
  X *_self; // might be different than XImpl::this
};
X::X()  { new (ximpl_) XImpl(this); }
X::~X() { (reinterpret_cast<XImpl*>(ximpl_)->~XImpl(); }
```

Template code blow

- Templates are instantiated on request
- Each different template argument type creates new specialization
  - Good: we can specialize for types
  - Bad: Code is generated for all different arguments
  - All template member functions are templates

Template code blow

```
template <class T>
class matrix
{
public:
    int get_cols() const { return cols_; }
    int get_rows() const { return rows_; }
private:
    int cols_;
    int rows_;
    T *elements_;
};

matrix<int>     mi;
matrix<double>  md;
matrix<long>    ml;
```

Template code blow

```
class matrix_base
{
public:
    int get_cols() const { return cols_; }
    int get_rows() const { return rows_; }
protected:
    int cols_;
    int rows_;
};

template <class T>
class matrix : public matrix_base
{
private:
    T *elements_;
};
```

Using C and C++ together

- Object model is (almost) the same
- C++ programs regularly call C libraries
- Issues:
  - Virtual inheritance
  - Virtual functions (pointer to vtbl)
  - Mangled name vs C linkage name

Using C and C++ together

```
//
// C++ header for C/C++ files:
//
#ifdef __cplusplus
extern "C"
{
#endif
    int     f(int);
    double  g(double, int);
    // …
#ifdef __cplusplus
}
#endif
```

Place of instantiation

```
#include <iostream>
#include <algorithm>

template<typename T>
struct Test{
     void operator()(T& lhs,T& rhs){
         std::swap(lhs,rhs);
     }
};

struct MyT { };

namespace std {
     inline void swap(MyT& lhs,MyT& rhs){
         std::cout << "MySwap" << std::endl;
     }
}

int main()
{
     MyT t1,t2;
     Test<MyT>{}(t1,t2);
}

$ ./a.out
$
```

Place of instantiation

```
#include <iostream>
#include <algorithm>

// move Test from here
struct MyT { };

namespace std {
     inline void swap(MyT& lhs,MyT& rhs){
         std::cout << "MySwap" << std::endl;
     }
}

// to here
template<typename T>
struct Test{
     void operator()(T& lhs,T& rhs){
         std::swap(lhs,rhs);
     }
};

int main()
{
     MyT t1,t2;
     Test<MyT>{}(t1,t2);
}

$ ./a.out
MySwap
$
```

Order of linking?

```
// file: a.cpp
static int s = 2;
#include "t.h"
int g()
{
    return t(1);
}
```

```
// file: b.cpp
static int s = 1;
#include "t.h"
int h()
{
    return t(1);
}
```

```
// file: t.h
template <class T>
T t(const T&)
{
    return s;
}
```

```
// file: main.cpp
#include <iostream>
extern int g();
extern int h();
int main()
{
    std::cout << g() <<std::endl;
    std::cout << h() <<std::endl;
    return 0;
}
```

```
$ g++ main.cpp a.cpp b.cpp  &&  ./a.out
2 2
```

Order of linking?

```
// file: a.cpp
static int s = 2;
#include "t.h"
int g()
{
    return t(1);
}
```

```
// file: b.cpp
static int s = 1;
#include "t.h"
int h()
{
    return t(1);
}
```

```
// file: t.h
template <class T>
T t(const T&)
{
    return s;
}
```

```
// file: main.cpp
#include <iostream>
extern int g();
extern int h();
int main()
{
    std::cout << g() <<std::endl;
    std::cout << h() <<std::endl;
    return 0;
}
```

```
$ g++ main.cpp b.cpp a.cpp  &&  ./a.out
1 1
```

Static initialization/destruction

- Static objects inside translation unit constructed in a well-defined order
- No ordering between translation units
- Issues:
  - (1) Constructor of static refers other source's static
  - Destruction order
- Lazy singleton solves (1)
- Phoenix pattern solves (2)