

Basic C++

Compiling & linking

Dr. Porkoláb Zoltán Károly

gsd@inf.elte.hu

<http://gsd.web.elte.hu>

Range issues

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstdlib>
#include <ranges>

void print( auto c)
{
    for ( auto [x,y] : c )
    {
        std::cout << x << ':' << y << ' ';
    }
    std::cout << '\n';
}

int main()
{
    std::vector<int> v(1000);
    std::generate( v.begin(), v.end(), []() { return std::rand(); });

    print( v | std::views::transform(
        [cnt=0](auto x) mutable { return std::pair{cnt++,x}; })
        | std::views::filter( [] (auto x) { return x.second % 2; } )
        | std::views::take(10) );

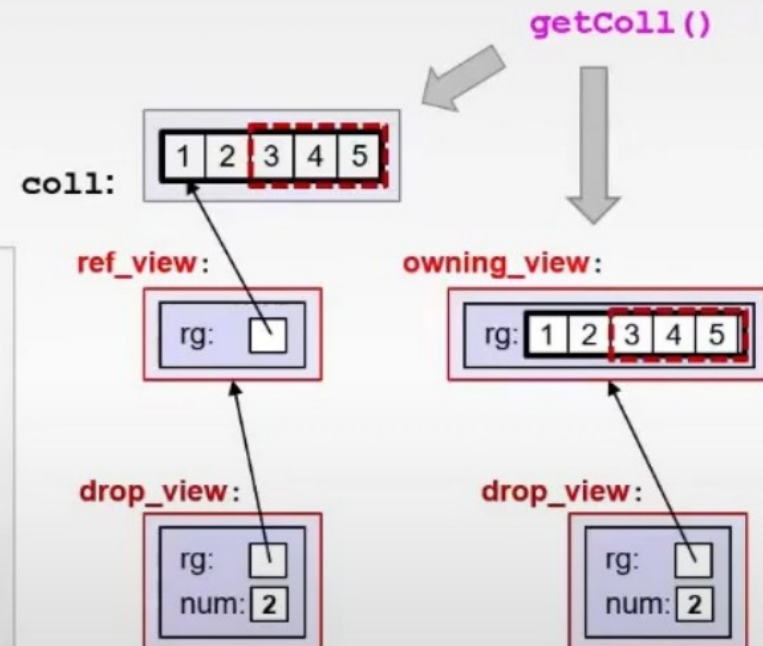
    return 0;
}
```

Range issues

- **ref_view** when referring to named objects (lvalues)
- **owning_view** when referring to temporaries (rvalues)

```
std::vector<int> getColl()  
{  
    return {1, 2, 3, 4, 5};  
}
```

```
auto coll = getColl();  
auto v1 = coll | std::views::drop(2);  
           drop_view<ref_view<vector<int>>>  
  
auto v2 = getColl() | std::views::drop(2);  
           drop_view<owning_view<vector<int>>>
```

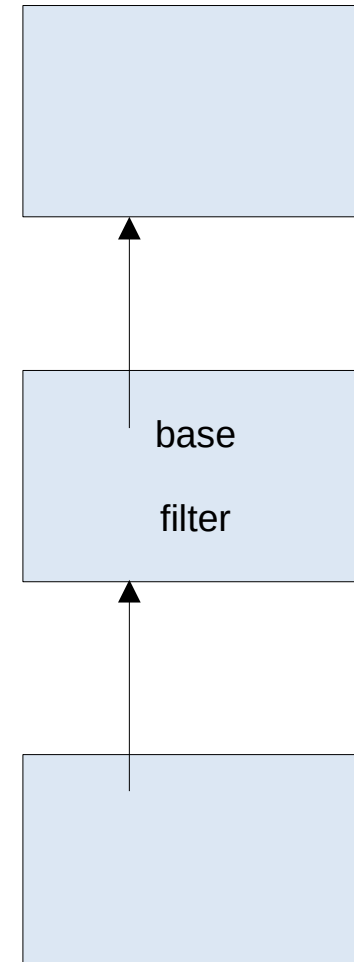


Range issues

```
// very high level view
struct filter_view_iterator
{
    auto operator*()
    {
        while ( ! func(*base) ) ++base;
        return *base;
    }

    Func    func;    // functor to filter
    Base    base;    // iterator to base item
}

```



Range issues

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstdlib>
#include <ranges>

void print( auto c)
{
    for ( auto [x,y] : c )
    {
        std::cout << x << ':' << y << ' ';
    }
    std::cout << '\n';
}

int main()
{
    std::vector<int> v(1000);
    std::generate( v.begin(), v.end(), []() { return std::rand(); });

    print( v | std::views::transform([cnt=-1, ptr=(int*)0] (auto& x) mutable {
        if( ptr != &x) {
            ptr = &x; ++cnt;
        }
        return std::pair{cnt,x};
    })
    | std::views::filter( [] (auto x) { return x.second % 2; } )
    | std::views::take(10) );

    return 0;
}
```

Constructor as conversion operator

```
class date { ... };
```

```
main()
```

```
{
```

```
    date exam{2019, 12, 15}; // constructor with 3 ints
```

```
    date semester{2022, 2}; // constructor with 2 ints
```

```
    if ( exam < date{2020} ) { ... } // 2020.1.1
```

```
    else if ( exam < date{} ) { ... } // today
```

```
    date exam2{"2019.12.15"}; // constructor with string
```

```
}
```

- How many constructors should we define?

Constructor as conversion operator

```
class date
{
public:
    date(); // default constructor
    date( int y, int m=1, int d=1 ); // initialiser constructor
    explicit date( const char *s ); // explicit constructor
    // ...
private:
    // ...
    int year;
    int month;
    int day;
};
```

Side note: Schwartz error

```
while ( cin >> i ) // 1st attempt: cin.operator int() Jerry Schwarz
```

```
std::istream& tempc = cin.operator>>(i);  
int tempi = tempc.operator int();  
while ( tempi )
```

```
while ( cin << i )  
  
int tempi1 = cin.operator int();  
int tempi2 = tempi1 << i;  
while ( tempi2 )
```

`operator bool()` is not better: `bool` -> `int` promotion

Side note: safe bool conversion

```
while ( cin >> i ) // 2nd attempt: cin.operator void*() Jerry Schwarz  
  
std::istream& tempc = cin.operator>>(i);  
void* tempptr = tempc.operator void*();  
while ( tempi ) // contextually converted to bool
```

```
while ( cin << i )  
  
void* tempi1 = cin.operator void*(); // no operator<<(void*,int)  
delete std::cin; // delete ( operator void*(std::cin) )
```

```
// C++11 solution: explicit specifier  
explicit operator bool() const; // not to use in implicit conversion  
// works in if, while, etc...
```

```
// C++20  
explicit (expr) operator bool() const; // explicit if expr is true
```

How to define operators?

- $a + b$

$a.\text{operator}+(b)$

$\text{operator}+(a,b)$

- $a = b$, $a[b]$, $a(b_1, b_2, \dots)$, $a->$ only member

$a.\text{operator}= (b)$

$a.\text{operator}[] (b)$

$a.\text{operator}() (b_1, b_2, \dots)$

$a.\text{operator}->()$

Where to define operators?

- Theory says: data and operations on it have strong binding
 - Member operators

- Some operators can't be members

```
std::ostream& operator<<(std::ostream&, const X&)
```

- Sometimes members creates unwanted dependencies

```
std::getline(std::basic_istream&, std::basic_string&)
```

- Sometime operators should be symmetric

Symmetry

```
class date
{
public:
    bool operator<(const date& rhs);
};

int main()
{
    date today;    // current date from OS

    if ( today < date(2016) )    // works

    if ( today < 2016 )          // works

};
```

Symmetry

```
class date
{
public:
    bool operator<(const date& rhs);
};

int main()
{
    date today;    // current date from OS

    if ( today < date(2016) ) // works

    if ( today < 2016 )        // works

    if ( 2016 < today )        // does not work
                                // if operator< is member
};
```

Symmetry

```
class date
{
public:
};

bool operator<(const date& lhs, const date& rhs);

int main()
{
    date today;    // current date from OS

    if ( today < date(2016) ) // works

    if ( today < 2016 )        // works

    if ( 2016 < today )        // works
                                // if operator< is global
};
```

Inline functions

```
// date.h
class date
{
public:
    // ...
private:
    // ...
    int year;
    int month;
    int day;
};

    bool operator< ( date d1, date d2 );
inline bool operator==( date d1, date d2 ) { ... }
inline bool operator!=( date d1, date d2 ) { ... }
inline bool operator<=( date d1, date d2 ) { ... }
inline bool operator>=( date d1, date d2 ) { ... }
inline bool operator> ( date d1, date d2 ) { ... }
```

Operators in C++20

```
#include <compare>
```

```
class date
```

```
{
```

```
public:
```

```
    // ...
```

```
    bool operator==(const Date& d) const = default;           // memberwise
```

```
    std::strong_ordering operator<=>(const Date& d) const;      // "spaceship"
```

```
private:
```

```
    // ...
```

```
    int year;
```

```
    int month;
```

```
    int day;
```

```
};
```

```
std::strong_ordering Date::operator<=>(const Date& d) const
```

```
{
```

```
    if (auto c = year_ <=> d.year_; c != 0) return c;
```

```
    if (auto c = month_ <=> d.month_; c != 0) return c;
```

```
    return day_ <=> d.day_;
```

```
}
```

```
// generated: bool Date::operator!=(const Date& d) { return !(*this == d); }
```

```
// generated: bool Date::operator<(const Date& d) { return (*this<=>d) < 0; }
```

```
// generated: bool Date::operator<=(const Date& d) { return (*this<=>d) <= 0; }
```

```
// etc...
```


Operators in C++20

```
#include <compare>
```

```
class date
```

```
{
```

```
public:
```

```
    // ...
```

```
    // bool operator==(const Date& d) const = default;
```

```
    std::strong_ordering operator<=>( const Date& d) const = default; // lexicogr
```

```
private:
```

```
    // ...
```

```
    int year;
```

```
    int month;
```

```
    int day;
```

```
};
```

```
std::strong_ordering Date::operator<=>( const Date& d) const
```

```
{
```

```
    if (auto c = year_ <=> d.year_; c != 0 ) return c;
```

```
    if (auto c = month_ <=> d.month_; c != 0 ) return c;
```

```
    return day_ <=> d.day_;
```

```
}
```

```
// generated: bool Date::operator==(const Date& d) { return (*this<=>d) == 0; }
```

```
// generated: bool Date::operator!=( const Date& d ) { return ! (*this == d); }
```

```
// generated: bool Date::operator<(const Date& d) { return (*this<=>d) < 0; }
```

```
// generated: bool Date::operator<=(const Date& d) { return (*this<=>d) <= 0; }
```

```
// etc...
```

Three way comparision

- Spaceship operator

$(a <=> b) < 0$ if $a < b$
 $(a <=> b) > 0$ if $a > b$
 $(a <=> b) == 0$ if $a == b$

- Return type

<code>std::strong_ordering</code>	equal, less, greater	$a == b \Rightarrow f(a) == f(b)$
<code>std::weak_ordering</code>	equivalent, less, greater	
<code>std::partial_ordering</code>	equivalent, less, greater, unordered	$1.f <=> \text{NaN}$

- Reversing operators

$a == 10$	$a.operator==(10)$	
$10 == a$	$a.operator==(10)$	assume symmetry
$a <=> 10$	$a.operator<=>(10)$	
$10 <=> a$	$a.operator<=>(10)$	assume symmetry

Three way comparision

- Spaceship operator

$(a <=> b) < 0$ if $a < b$
 $(a <=> b) > 0$ if $a > b$
 $(a <=> b) == 0$ if $a == b$

- Return type

<code>std::strong_ordering</code>	equal, less, greater	$a == b \Rightarrow f(a) == f(b)$
<code>std::weak_ordering</code>	equivalent, less, greater	
<code>std::partial_ordering</code>	equivalent, less, greater, unordered	$1.f <=> \text{NaN}$

- Reversing operators

<code>a == 10</code>	<code>a.operator==(10)</code>
<code>10 == a</code>	<code>a.operator==(10)</code> assume symmetry
<code>a <=> 10</code>	<code>a.operator<=>(10)</code>
<code>10 <=> a</code>	<code>a.operator<=>(10)</code> assume symmetry
<code>10 <=> a</code>	<code>0 <=> a.operator<=>(10)</code> assume symmetry

Partial ordering

```
#include <optional>
#include <compare>
```

<https://brevzin.github.io/c++/2019/07/28/comparisons-cpp20/>

```
struct IntNan {
    std::optional<int> val = std::nullopt;

    bool operator==(IntNan const& rhs) const {
        if (!val || !rhs.val) {
            return false;
        }
        return *val == *rhs.val;
    }
    std::partial_ordering operator<=>(IntNan const& rhs) const {
        if (!val || !rhs.val) {
            // we can express the unordered state as a first class value
            return std::partial_ordering::unordered;
        }
        // int <=> int returns strong_ordering, convertible to partial_ordering
        return *val <=> *rhs.val;
    }
};

// (IntNan{2} <=> IntNan{4}) < 0 ) // true
// (IntNan{2} <=> IntNan{}) == std::partial_ordering::unordered // true
// IntNan{2} <= IntNan{} // false
```

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
- Necessary 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

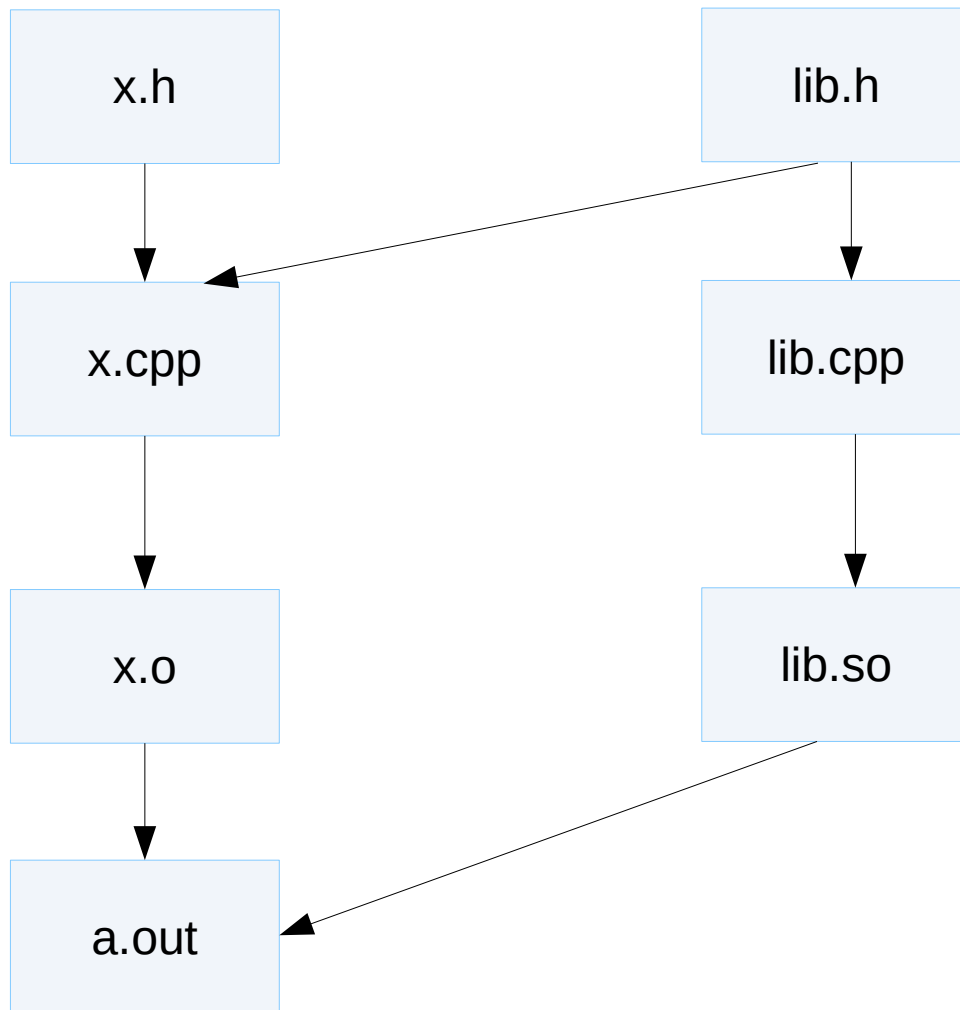
- ```
#include <iosfwd>
#include <list>
// none of A, B, C, D, E are templates
// Only A and C have virtual functions

#include "a.h" // class A
#include "b.h" // class B
// #include "c.h" // class C DEPENDS!
#include "d.h" // class D

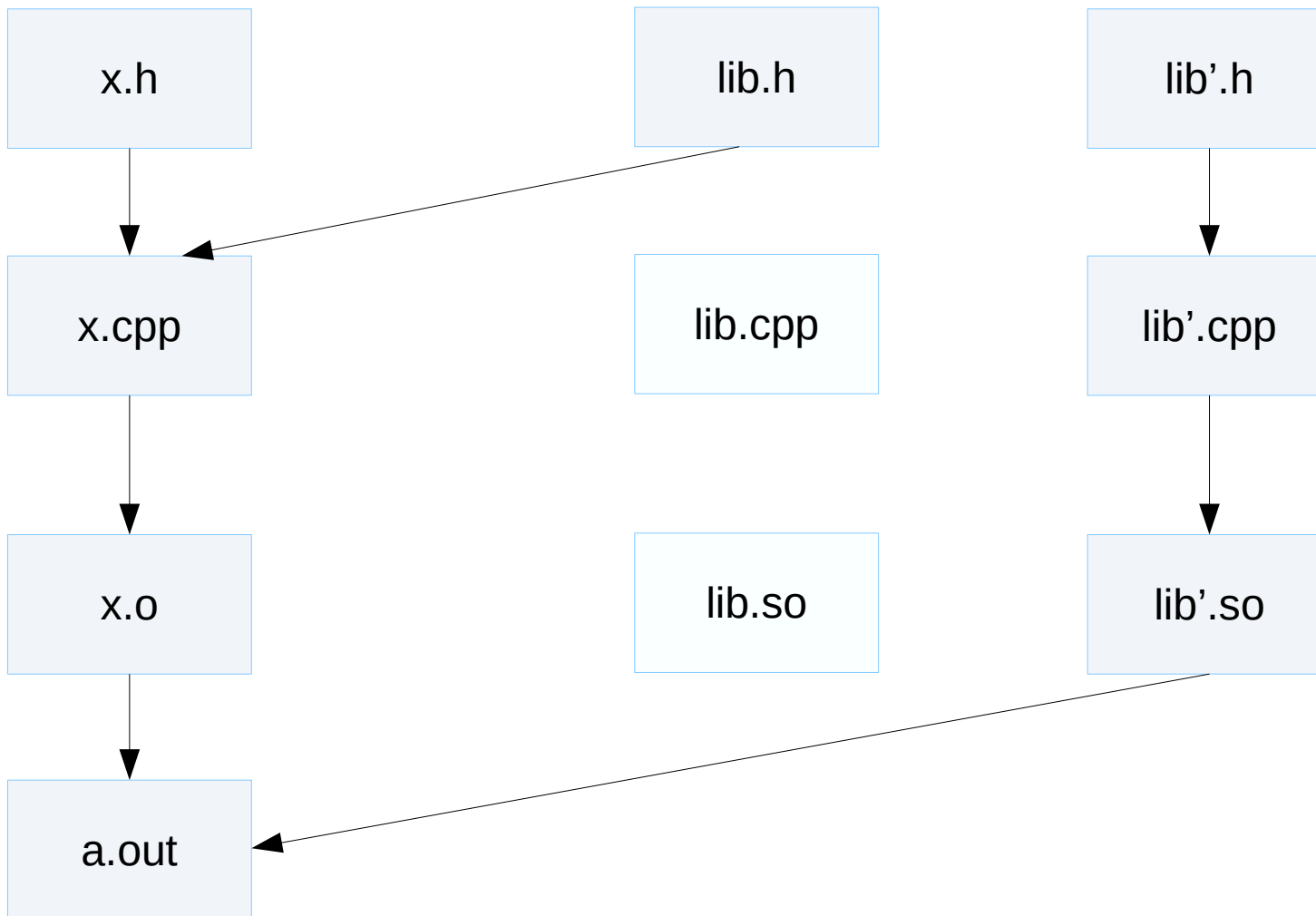
class C; class E; // forward declaration

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_;
};
```

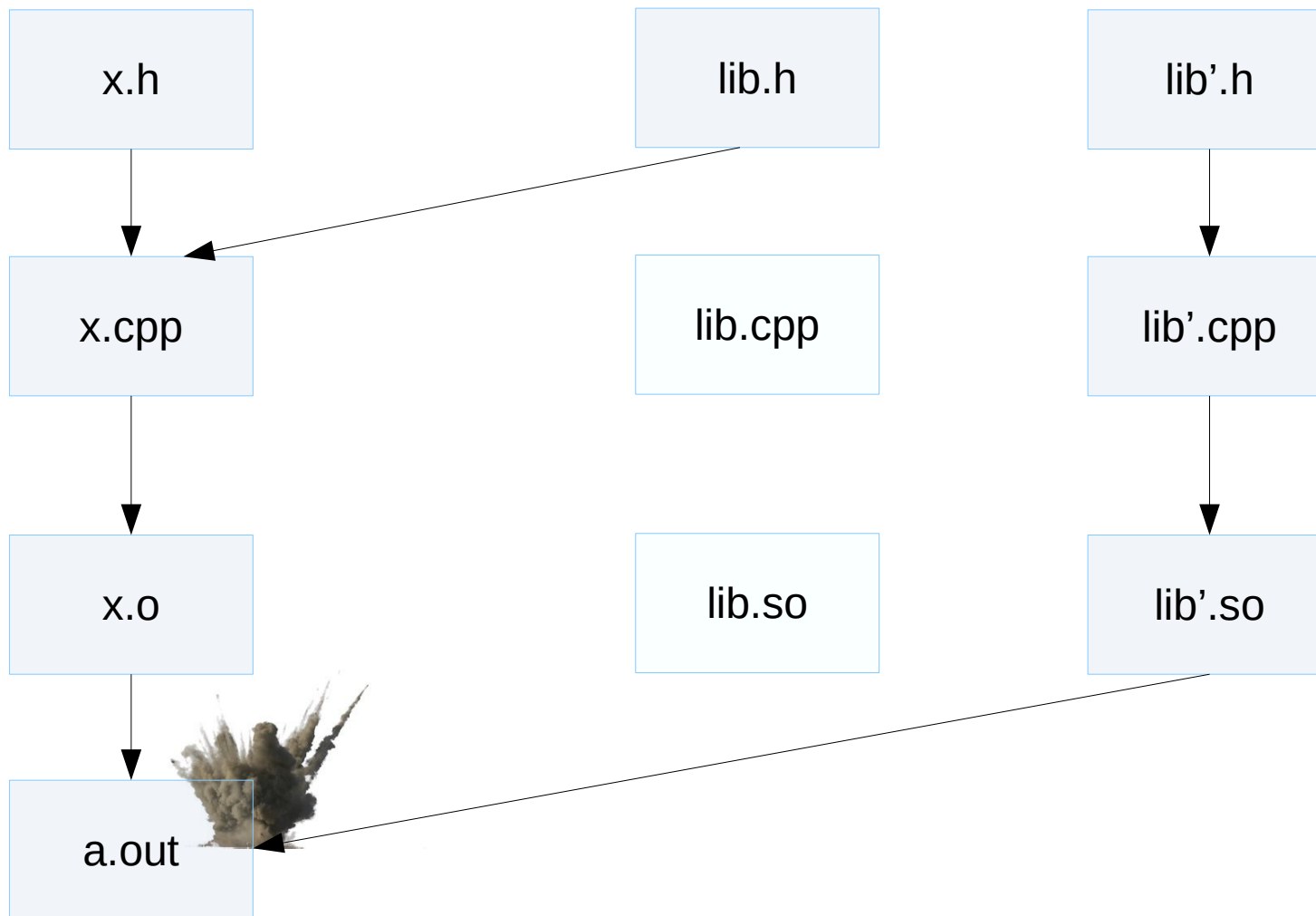
# Binary compatibility



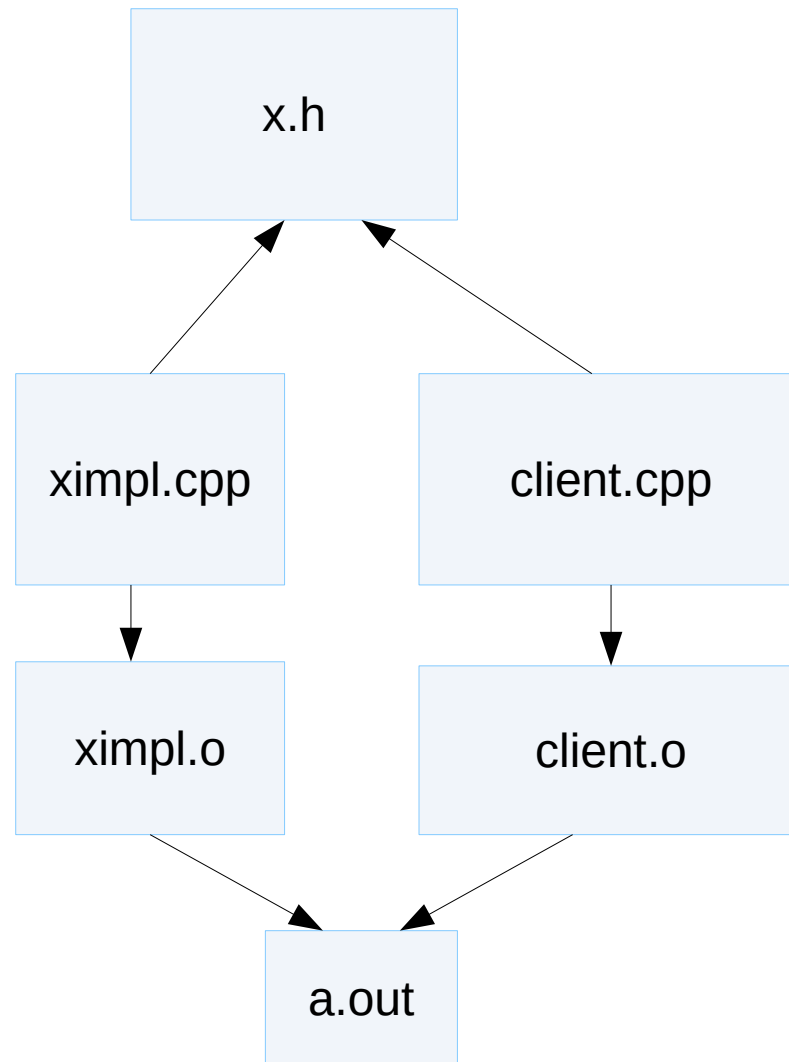
# Binary compatibility



# Binary compatibility



# PIMPL



# PIMPL

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

```
// file ximpl.cpp
struct XImpl // not necessary to declare as "class"
{
 // private members; fully hidden
 // can be changed 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
#include <list>
struct Ximpl
{
 std::list<C> clist_; // class list<C> and D is hidden
 D d_;
};
```

# PIMPL with unique\_ptr

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

```
// file ximpl.cpp
struct X::XImpl // not necessary to declare as "class"
{
 XImpl() { ... }
 // can be changed 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() { } compiler generated
private:
    // pointer to forward declared class
    struct XImpl;
    std::unique_ptr<XImpl> pimpl_; // opaque pointer
};

// file ximpl.cpp
struct X::XImpl // not necessary to declare as "class"
{
    XImpl() { ... }
    // can be changed without recompiling clients
};

X::X() : pimpl_(std::make_unique<XImpl>()) {}
```

PIMPL with unique_ptr

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

// file ximpl.cpp
struct X::XImpl // not necessary to declare as "class"
{
 XImpl() { ... }
 // can be changed 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;
 std::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;
 std::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
#include <list>
struct Ximpl
{
 B b_; // delegate
 std::list<C> clist_;
 D d_;
};
```

# Fast PIMPL

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

// file ximpl.cpp
struct XImpl // not necessary to declare as "class"
{
 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 necessary to declare as "class"
{
 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>

// which swap?
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>

// from here
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;
    }
}
// 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
$
```

Correct use of swap

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

struct MyT { };

namespace std { // do not extend std namespace
    inline void swap(MyT& lhs, MyT& rhs) {
        std::cout << "MySwap" << std::endl;
    }
}
// works here
template<typename T>
struct Test {
    void operator()(T& lhs, T& rhs) {
        using std::swap; // introduce std::swap into the scope
        swap(lhs, rhs); // use ADL to select the better swap
    }
};

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

$ ./a.out
MySwap
$
```


Correct use of swap

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

struct MyT { };

namespace std { // do not extend std namespace
    inline void swap(MyT& lhs, MyT& rhs) {
        std::cout << "MySwap" << std::endl;
    }
}
// works here
template<typename T>
struct Test {
    void operator()(T& lhs, T& rhs) {
        using std::swap; // introduce std::swap into the scope
        swap(lhs, rhs); // use ADL to select the better swap
    }
};
inline void swap(MyT& lhs, MyT& rhs) {
    std::cout << "MySwap" << std::endl;
}

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

$ ./a.out
MySwap
$
```

to_string example

```
#include <string>

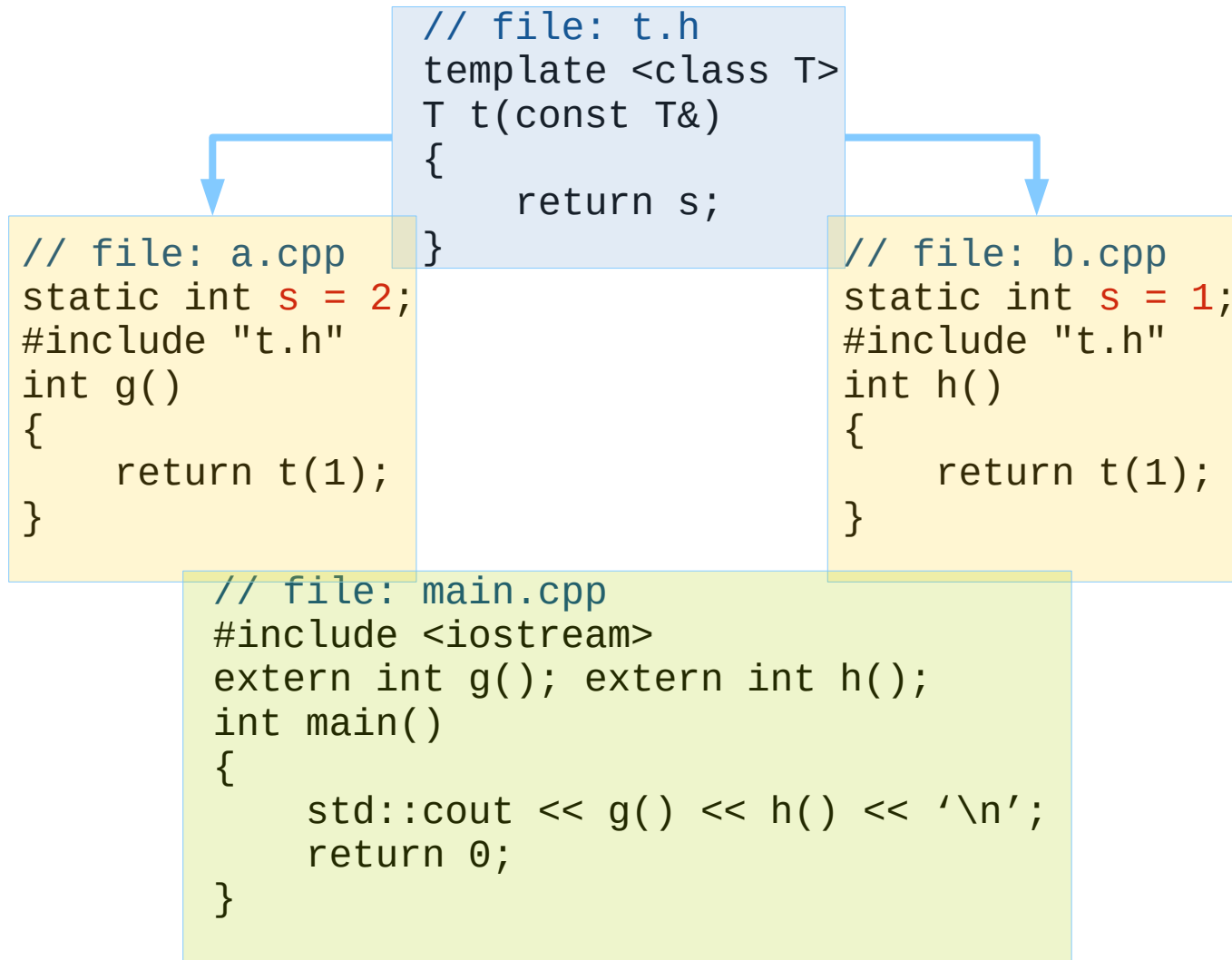
template <typename T>
class MyClass
{
public:
    std::string to_string() const;
private:
    std::string to_string_ads(const T &t) const;
};

template <typename T>
std::string MyClass<T>::to_string_ads(const T &t) const // ADS helper
{
    using std::to_string;
    return to_string(t);
}

int main()
{
    MyClass<double> td;
    MyClass<X>      tx;

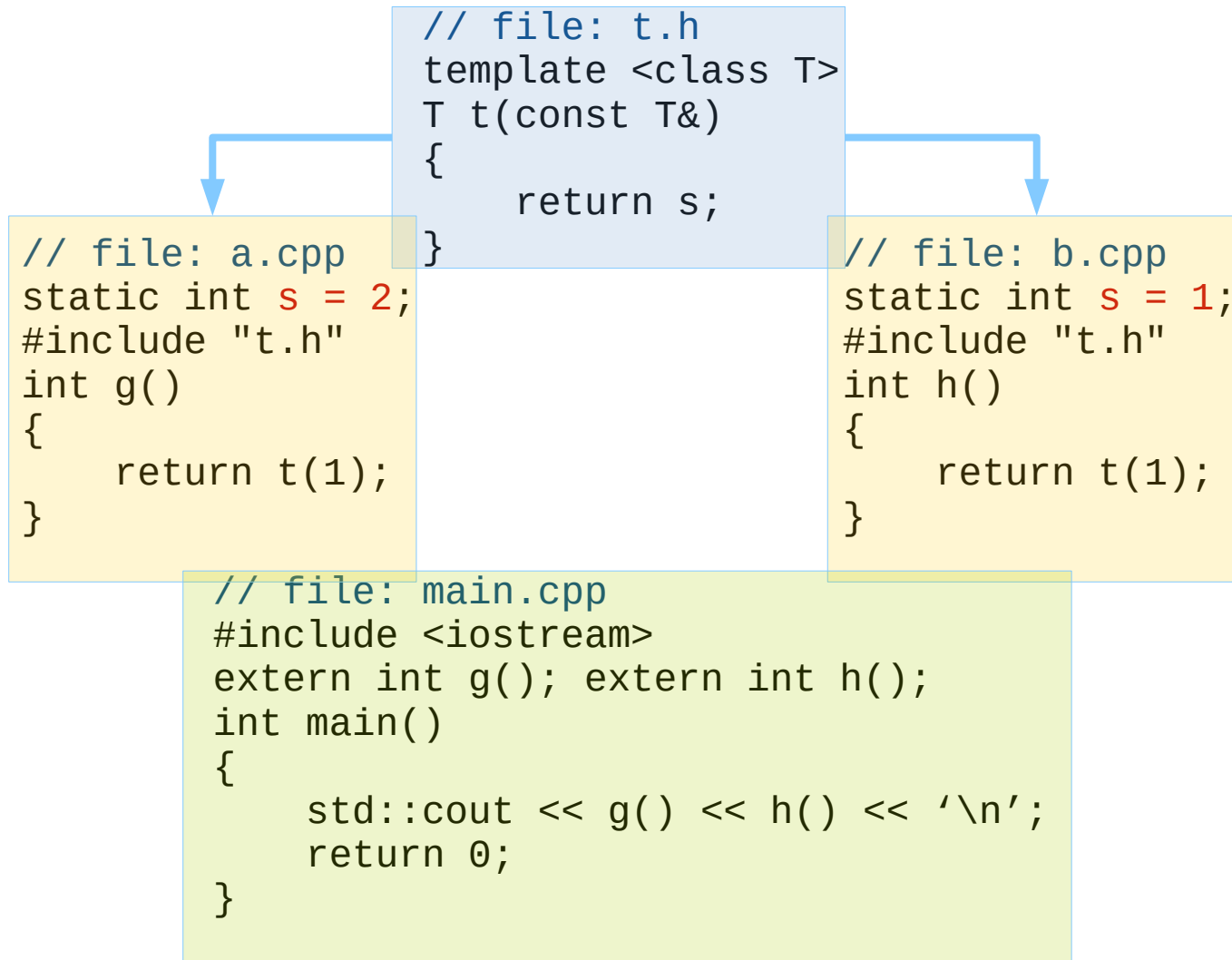
    td.to_string(); // calls std::to_string(double);
    tx.to_string(); // calls to_string(const X&);
}
```

Order of linking?



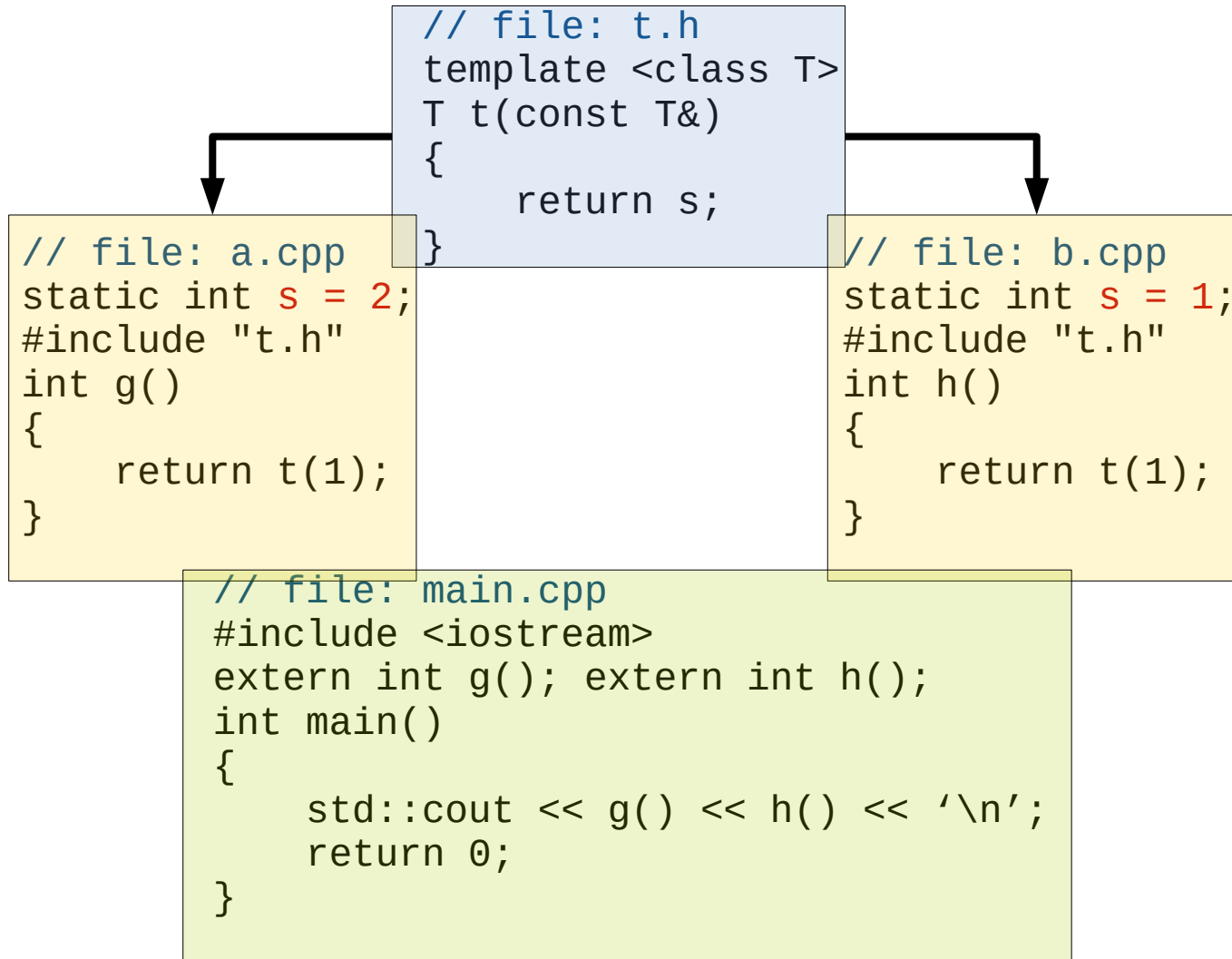
```
$ g++ main.cpp a.cpp b.cpp && ./a.out
```

Order of linking?



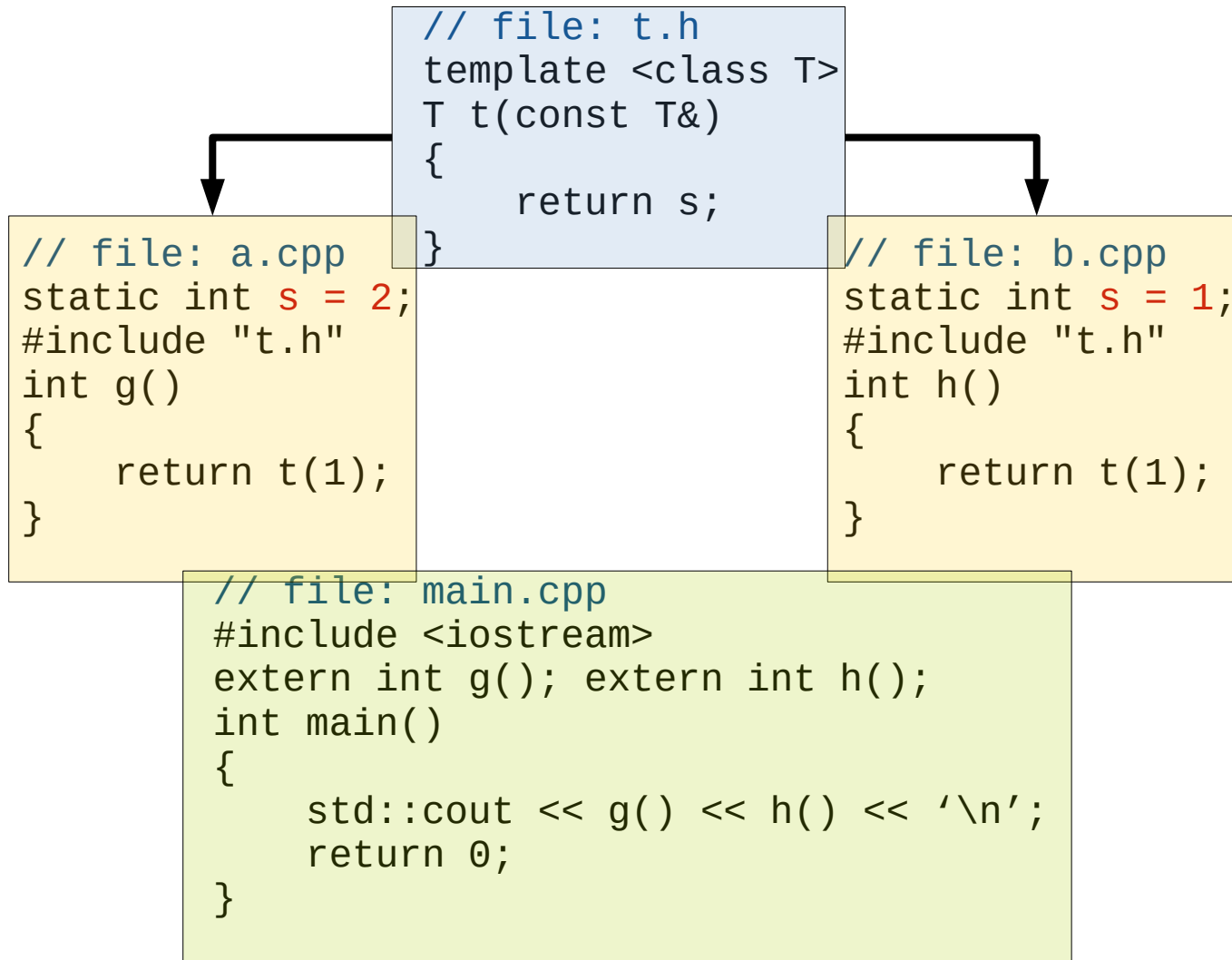
```
$ g++ main.cpp a.cpp b.cpp && ./a.out
2 2
```

Order of linking?



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

Order of linking?



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

```
1 1
```

Inline functions

```
// Date.h
class Date
{
public:
    // ...
private:
    // ...
    int year;
    int month;
    int day;
};

// operators for class Date
inline bool operator<( date d1, date d2) { ... }
inline bool operator==(date d1,date d2) { ... }
inline bool operator!=( date d1, date d2) { ... }
inline bool operator<=( date d1, date d2) { ... }
inline bool operator>=( date d1, date d2) { ... }
inline bool operator>( date d1, date d2) { ... }
```

Inline variables C++17

```
#include <iostream>
void f();

    int g0 = 10;
    inline int g1 = 11;
    static inline int g2 = 12;
    extern inline int g3 = 13;

int main()
{
    std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
    std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
    std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
    f();
    return 0;
}
```

```
#include <iostream>

    int g0 = 20;
    inline int g1 = 21;
    static inline int g2 = 22;
    extern inline int g3 = 23;

void f()
{
    std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
    std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
    std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
}
```


Inline variables C++17

```
#include <iostream>
void f();
```

```
    int g0 = 10;
```

```
    inline int g1 = 11;
```

```
    static inline int g2 = 12;
```

```
    extern inline int g3 = 13;
```

```
int main()
```

```
{
```

```
    std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
```

```
    std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
```

```
    std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
```

```
    f();
```

```
    return 0;
```

```
}
```

/usr/bin/ld: inline2.o:(.data+0x0): multiple definition of `g0'; inline1.o:(.data+0x0): first defined here

```
#include <iostream>
```

```
    int g0 = 20;
```

```
    inline int g1 = 21;
```

```
    static inline int g2 = 22;
```

```
    extern inline int g3 = 23;
```

```
void f()
```

```
{
```

```
    std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
```

```
    std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
```

```
    std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
```

```
}
```

Inline variables C++17

```
#include <iostream>
void f();
```

```
    int g0 = 10;
```

```
    inline int g1 = 11;
```

```
    static inline int g2 = 12;
```

```
    extern inline int g3 = 13;
```

```
int main()
```

```
{
```

```
    std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
```

```
    std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
```

```
    std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
```

```
    f();
```

```
    return 0;
```

```
}
```

/usr/bin/ld: inline2.o:(.data+0x0): multiple definition of `g0'; inline1.o:(.data+0x0): first defined here

```
#include <iostream>
```

```
    extern int g0 = 20;
```

```
    inline int g1 = 21;
```

```
    static inline int g2 = 22;
```

```
    extern inline int g3 = 23;
```

```
void f()
```

```
{
```

```
    std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
```

```
    std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
```

```
    std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
```

```
}
```

Inline variables C++17

```
#include <iostream>
void f();

    int g0 = 10;
    inline int g1 = 11;
    static inline int g2 = 12;
    extern inline int g3 = 13;

int main()
{
    std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
    std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
    std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
    f();
    return 0;
}
```

```
#include <iostream>
    extern int g0;
    inline int g1 = 21;
    static inline int g2 = 22;
    extern inline int g3 = 23;

void f()
{
    std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
    std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
    std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
}
```

Inline variables C++17

```
#include <iostream>
void f();
    int g0 = 10;
    inline int g1 = 11;
    static inline int g2 = 12;
    extern inline int g3 = 13;

int main()
{
    std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
    std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
    std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
    f();
    return 0;
}
```

```
&g0 = 10, 0x40406c
&g1 = 21, 0x404060
&g2 = 12, 0x404070
&g3 = 23, 0x404068
&g0 = 10, 0x40406c
&g1 = 21, 0x404060
&g2 = 22, 0x404064
&g3 = 23, 0x404068
```

```
#include <iostream>
    extern int g0;
    inline int g1 = 21;
    static inline int g2 = 22;
    extern inline int g3 = 23;

void f()
{
    std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
    std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
    std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
}
```

Inline variables C++17

```
#include <iostream>
void f();
    int g0 = 10;
    inline int g1 = 11;
    static inline int g2 = 12;
    extern inline int g3 = 13;

int main()
{
    std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
    std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
    std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
    f();
    return 0;
}
```

```
&g0 = 10, 0x40406c
&g1 = 11, 0x404060
&g2 = 12, 0x404070
&g3 = 13, 0x404068
&g0 = 10, 0x40406c
&g1 = 11, 0x404060
&g2 = 22, 0x404064
&g3 = 13, 0x404068
```

```
#include <iostream>
    extern int g0;
    inline int g1 = 21;
    static inline int g2 = 22;
    extern inline int g3 = 23;

void f()
{
    std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
    std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
    std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
}
```

Linker symbols

```
$ readelf inline1.o
```

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	inline1.cpp
2:	0000000000000004	11	FUNC	LOCAL	DEFAULT	4	_GLOBAL__sub_I_inline1.cp
3:	0000000000000004	4	OBJECT	LOCAL	DEFAULT	7	_ZL2g2
4:	0000000000000000	1	OBJECT	LOCAL	DEFAULT	6	_ZStL8__ioinit
5:	0000000000000000	59	FUNC	LOCAL	DEFAULT	4	__cxx_global_var_init
6:	0000000000000000	0	SECTION	LOCAL	DEFAULT	2	
7:	0000000000000000	0	SECTION	LOCAL	DEFAULT	4	
8:	0000000000000000	0	SECTION	LOCAL	DEFAULT	6	
9:	0000000000000000	0	SECTION	LOCAL	DEFAULT	7	
10:	0000000000000000	0	SECTION	LOCAL	DEFAULT	8	
11:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	_Z1fv
12:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	_ZNSolsEPKv
13:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	_ZNSolsEi
14:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	_ZNSt8ios_base4InitC1Ev
15:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	_ZNSt8ios_base4InitD1Ev
16:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	_ZSt4cout
17:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	_ZStlsISt11char_traitsIcE
18:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	_ZStlsISt11char_traitsIcE
19:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	__cxa_atexit
20:	0000000000000000	0	NOTYPE	GLOBAL	HIDDEN	UND	__dso_handle
21:	0000000000000000	4	OBJECT	GLOBAL	DEFAULT	7	g0
22:	0000000000000000	4	OBJECT	WEAK	DEFAULT	10	g1
23:	0000000000000000	4	OBJECT	WEAK	DEFAULT	12	g3
24:	0000000000000000	316	FUNC	GLOBAL	DEFAULT	2	main

Static initialization/destruction

- Matt Godbolt cppcon 18 The Bits between the Bits
<https://youtu.be/dOfucXtyEsU>

```
typedef void (*init_func)(int, char **, char **);

extern init_func __init_array_start[];
extern init_func __init_array_end[];

int __libc_csu_init(int argc, char **argv, char **envp) {
    const size_t size = __init_array_end - __init_array_start;
    for (size_t i = 0; i < size; i++)
        (*__init_array_start[i])(argc, argv, envp);
}
```

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
 - (2) Destruction order
- Lazy singleton solves (1)
- Schwartz counter solves (1+2)

Schwartz counter

```
// init.h
class InitMngr
{
public:
    InitMngr() { if ( !count_++ ) init(); }
    ~InitMngr() { if ( !--count_ ) cleanup(); }
    void init();
    void cleanup();
private:
    static long count_; // one per process
};
namespace { InitMngr initMngr; } // one per file inclusion

// init.cpp
long InitMngr::count_ = 0;
void InitMngr::init() { /* initialization */ }
void InitMngr::cleanup() { /* cleanup */ }
```