Back To Basics
Friendship

MATEUSZ PUSZ

Workshopy Style

Workshopy Style

- Provide rationale
- Facilitate discussion
  - force the audience to think
  - not just a lecture
- Describe
  - pitfalls
  - corner cases
- Provide recommendations
- ~~Lot's of coding~~

https://ahaslides.com/RX45U

Poll

Did you use **`friend`** keyword at least once in your project?

The Art of C++ Friendship

`friend` is a powerful tool, but like art, it requires skill, understanding, and careful application.

The Art of C++ Friendship

`friend` is a powerful tool, but like art, it requires skill, understanding, and careful application.

Avoid overuse, use it judiciously, and appreciate its nuances.

"Friend is evil": Myth or Truth?

Common belief: `friend` breaks encapsulation and should be avoided

"Friend is evil": Myth or Truth?

Common belief: `friend` breaks encapsulation and should be avoided

Let's challenge some assumptions 🤔

"Friend is evil": Myth or Truth?

Common belief: `friend` breaks encapsulation and should be avoided

Reality check: We'll discover that:

- `friend` is often **misunderstood** rather than inherently bad
- **Hidden friends** can actually improve your code quality
- The **real problems** come from misuse, not the feature itself
- **Modern C++** has elegant patterns that make `friend` shine

Quiz: What will be the outcome of this program?

```cpp
#include <iostream>

struct my_int {
  int value_;
public:
  constexpr my_int(int value): value_(value) {}
};

int main()
{
  std::cout << my_int{42} << '\n';
}
```

Quiz: What will be the outcome of this program?

```cpp
#include <iostream>

struct my_int {
  int value_;
public:
  constexpr my_int(int value): value_(value) {}
};

int main()
{
  std::cout << my_int{42} << '\n';
}
```

```
error: invalid operands to binary expression ('ostream' (aka 'basic_ostream<char>') and 'my_int')
   11 |   std::cout << my_int{42} << '\n';
      |   ~~~~~~~~~ ^  ~~~~~~~~~~~
```

Customization Points

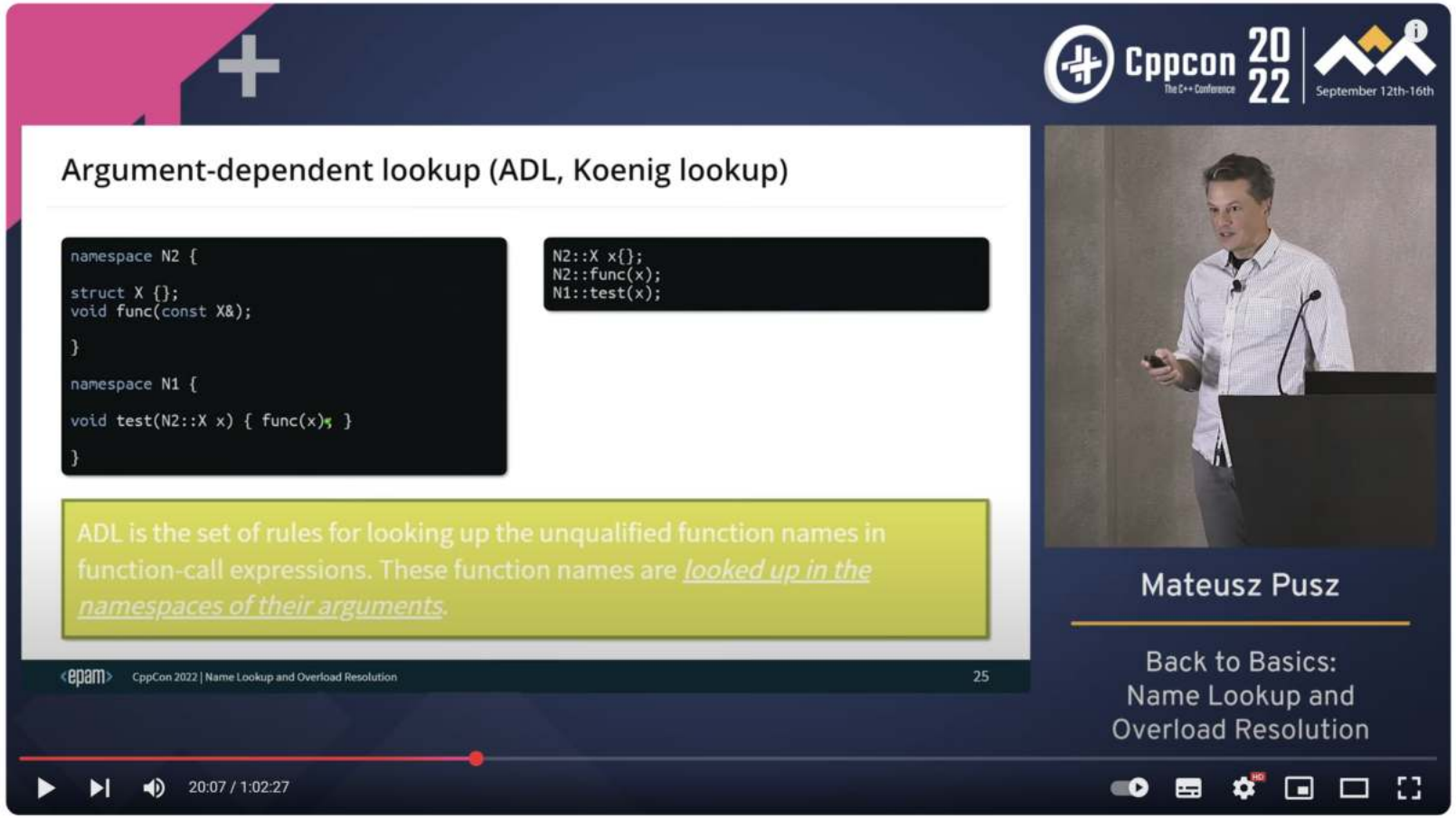
`operator<<` is a customization point that allows fundamental and user defined types to be inserted into the output stream.

Customization Points

`operator<<` is a customization point that allows fundamental and user defined types to be inserted into the output stream.

Most generic frameworks depend on the Argument Dependent Lookup (ADL) to find functions that customize behavior for a specific type.

Back To Basics: Name Lookup and Overload Resolution in C++



Back to Basics - Name Lookup and Overload Resolution in C++ - Mateusz Pusz - CppCon 2022

Quiz: What will be the outcome of this program?

```cpp
struct my_int {
  int value_;
public:
  constexpr my_int(int value): value_(value) {}
  // ...
};

std::ostream& operator<<(std::ostream& os, my_int mi)
{
  return os << mi.value_;
}
```

```cpp
my_int i = 42;
std::cout << i << '\n';
```

Quiz: What will be the outcome of this program?

```cpp
struct my_int {
  int value_;
public:
  constexpr my_int(int value): value_(value) {}
  // ...
};

std::ostream& operator<<(std::ostream& os, my_int mi)
{
  return os << mi.value_;
}
```

```cpp
my_int i = 42;
std::cout << i << '\n';
```

42

Class Members Access Control

```cpp
class my_int {
  int value_;
public:
  constexpr my_int(int value): value_(value) {}
  // ...
};

std::ostream& operator<<(std::ostream& os, my_int mi)
{
  return os << mi.value_;
}
```

```cpp
my_int i = 42;
std::cout << i << '\n';
```

```
error: 'int my_int::value_' is private within this context
   12 |   return os << mi.value_;
      |                    ^~~~~~
note: declared private here
    4 |   int value_;
      |       ^~~~~~
```

Class Members Access Control

C++ offers a rich set of access specifiers to control the visibility of class members.

Class Members Access Control

C++ offers a rich set of access specifiers to control the visibility of class members.

`public` MEMBERS

- Accessible by everyone

Class Members Access Control

C++ offers a rich set of access specifiers to control the visibility of class members.

`public` MEMBERS

- Accessible by everyone

`protected` MEMBERS

- Accessible from the current class and its children

Class Members Access Control

C++ offers a rich set of access specifiers to control the visibility of class members.

`public` MEMBERS

- Accessible by everyone

`protected` MEMBERS

- Accessible from the current class and its children

`private` MEMBERS

- Accessible only from the current class

Default Class Members Access

`class`

- **`private`** access to members
- **`private`** inheritance

```
class Derived : public Base {
  int member_;
public:
  // public interface...
};
```

Default Class Members Access

`class`

- **`private`** access to members
- **`private`** inheritance

```
class Derived : public Base {
  int member_;
public:
  // public interface...
};
```

`struct`

- **`public`** access to members
- **`public`** inheritance

```
struct Derived : Base {
  // public interface...
private:
  int member;
};
```

Getters Are Often Not The Solution

```
class my_int {
  int value_;
public:
  constexpr my_int(int value): value_(value) {}

  constexpr int value() const { return value_; }
  // ...
};

std::ostream& operator<<(std::ostream& os,
                         my_int mi)
{
  return os << mi.value();
}
```

```
my_int i = 42;
std::cout << i << '\n';
```

42

Getters Are Often Not The Solution

Adding a getter may work, but it often breaks encapsulation by surfacing the implementation details to the user.

Quiz: What is the result of this code?

```
class my_int {
  int value_;
public:
  constexpr my_int(int value): value_(value) {}

  constexpr my_int operator+(my_int other) const
  {
    return value_ + other.value_;
  }
  // ...
};
```

```
my_int i = 42;
std::cout << i + 1 << '\n';   // #1
std::cout << 1 + i << '\n';   // #2
```

Quiz: What is the result of this code?

```cpp
class my_int {
  int value_;
public:
  constexpr my_int(int value): value_(value) {}

  constexpr my_int operator+(my_int other) const
  {
    return value_ + other.value_;
  }
  // ...
};
```

```cpp
my_int i = 42;
std::cout << i + 1 << '\n';   // #1
std::cout << 1 + i << '\n';   // #2
```

```
error: no match for 'operator+' (operand types are 'int' and 'my_int')
   18 |   std::cout << 1 + i << '\n';   // #2
      |                ~ ^ ~
      |                |   |
      |                int my_int
```

Overloading Binary Operators

Binary operators should typically be overloaded as **non-member functions** which allows the implicit conversions (if any) for both of the arguments.

Overloading Binary Operators

```
class my_int {
  int value_;
public:
  constexpr my_int(int value): value_(value) {}
  // ...
};

constexpr my_int operator+(my_int lhs, my_int rhs)
{
  return lhs.value() + rhs.value();
}
```

```
my_int i = 42;
std::cout << i + 1 << '\n';   // #1
std::cout << 1 + i << '\n';   // #2
```

Overloading Binary Operators

```
class my_int {
  int value_;
public:
  constexpr my_int(int value): value_(value) {}
  // ...
};

constexpr my_int operator+(my_int lhs, my_int rhs)
{
  return lhs.value() + rhs.value();
}
```

```
my_int i = 42;
std::cout << i + 1 << '\n';   // #1
std::cout << 1 + i << '\n';   // #2
```

```
43
43
```

Overloading Binary Operators

```
class my_int {
  int value_;
public:
  constexpr my_int(int value): value_(value) {}
  // ...
};

constexpr my_int operator+(my_int lhs, my_int rhs)
{
  return lhs += rhs;
}
```

```
my_int i = 42;
std::cout << i + 1 << '\n';   // #1
std::cout << 1 + i << '\n';   // #2
```

```
43
43
```

Why Does `friend` Exist?

Access control is a key principle in C++.

Why Does `friend` Exist?

Access control is a key principle in C++.

Some non-member functions **need** access to class private members. Many of them **can't be implemented as member** functions.

Why Does `friend` Exist?

```cpp
class my_int {
  int value_;
public:
  constexpr my_int(int value): value_(value) {}

  // Granting access
  friend std::ostream& operator<<(std::ostream&, my_int);

  friend constexpr my_int operator+(my_int, my_int);
  // ...
};
```

Why Does `friend` Exist?

```cpp
class my_int {
  int value_;
public:
  constexpr my_int(int value): value_(value) {}

  // Granting access
  friend std::ostream& operator<<(std::ostream&, my_int);

  friend constexpr my_int operator+(my_int, my_int);
  // ...
};
```

```cpp
std::ostream& operator<<(std::ostream& os, my_int mi)
{ return os << mi.value_; }

constexpr my_int operator+(my_int lhs, my_int rhs)
{ return lhs.value_ + rhs.value_; }
```

Why Does `friend` Exist?

```cpp
class my_int {
  int value_;
public:
  constexpr my_int(int value): value_(value) {}

  // Granting access
  friend std::ostream& operator<<(std::ostream&, my_int);

  friend constexpr my_int operator+(my_int, my_int);
  // ...
};
```

```cpp
std::ostream& operator<<(std::ostream& os, my_int mi)
{ return os << mi.value_; }

constexpr my_int operator+(my_int lhs, my_int rhs)
{ return lhs.value_ + rhs.value_; }
```

```cpp
my_int i = 42;
std::cout << i + 1 << '\n';
std::cout << 1 + i << '\n';
```

```
43
43
```

What Is `friend`?

Grants a function or class access to private/protected members of another class.

What Is `friend`?

Grants a function or class access to private/protected members of another class.

Traditionally used when external functions or classes **need a direct access** to class members.

Quiz: Is there any difference?

```cpp
class my_int {
  int value_;
public:
  constexpr my_int(int value): value_(value) {}

  // Granting access
  friend std::ostream& operator<<(std::ostream&, my_int);

  friend constexpr my_int operator+(my_int, my_int);
  // ...
};
```

```cpp
class my_int {
  int value_;

  // Granting access
  friend std::ostream& operator<<(std::ostream&, my_int);

  friend constexpr my_int operator+(my_int, my_int);
public:
  constexpr my_int(int value): value_(value) {}
  // ...
};
```

Quiz: Is there any difference?

```cpp
class my_int {
  int value_;
public:
  constexpr my_int(int value): value_(value) {}

  // Granting access
  friend std::ostream& operator<<(std::ostream&, my_int);

  friend constexpr my_int operator+(my_int, my_int);
  // ...
};
```

```cpp
my_int i = 42;
std::cout << i + 1 << '\n';
std::cout << 1 + i << '\n';
```

```cpp
class my_int {
  int value_;

  // Granting access
  friend std::ostream& operator<<(std::ostream&, my_int);

  friend constexpr my_int operator+(my_int, my_int);
public:
  constexpr my_int(int value): value_(value) {}
  // ...
};
```

```
43
43
```

`friend` Vs Access Specifiers

Access specifiers have no effect on the meaning of `friend` declarations.

`friend` Vs Access Specifiers

Access specifiers have no effect on the meaning of **`friend`** declarations.

`friend` declarations can appear in **`private`** or in **`public`** sections, with no difference.

Our Friends

- **Functions**
  - *non-member* functions
  - *member* functions of another class

Our Friends

- **Functions**
  - *non-member* functions
  - *member* functions of another class
- **Classes**

Our Friends

- **Functions**
  - *non-member* functions
  - *member* functions of another class
- **Classes**
- **Templates**
  - *non-member* function template
  - *member* function template
  - *class* template

Friend Non-Member Functions

```
class bank_account {
  int balance_;
  friend bool transfer_funds(bank_account& from, bank_account& to, int amount);
public:
  explicit bank_account(int balance) : balance_{balance} {}
  int balance() const { return balance_; }
};
```

Friend Non-Member Functions

```cpp
class bank_account {
  int balance_;
  friend bool transfer_funds(bank_account& from, bank_account& to, int amount);
public:
  explicit bank_account(int balance) : balance_{balance} {}
  int balance() const { return balance_; }
};
```

```cpp
bool transfer_funds(bank_account& from, bank_account& to, int amount)
{
  if (from.balance() < amount)
    return false;
  from.balance_ -= amount;
  to.balance_ += amount;
  return true;
}
```

Quiz: Friend Member Functions

```
class bank_account;

class transaction : nonmovable {
  virtual bool check_balance(const bank_account& from, int amount);
  virtual void transfer(bank_account& from, bank_account& to, int amount);
public:
  virtual ~transaction() {}
  bool run(bank_account& from, bank_account& to, int amount)
  { return check_balance(from, amount) ? transfer(from, to, amount), true : false; }
};
```

```
class bank_account {
  int balance_;
  friend void transaction::transfer(bank_account& from, bank_account& to, int amount);
public:
  explicit bank_account(int balance) : balance_{balance} {}
  int balance() const { return balance_; }
};
```

Quiz: Friend Member Functions

```
class bank_account;

class transaction : nonmovable {
  virtual bool check_balance(const bank_account& from, int amount);
  virtual void transfer(bank_account& from, bank_account& to, int amount);
public:
  virtual ~transaction() {}
  bool run(bank_account& from, bank_account& to, int amount)
  { return check_balance(from, amount) ? transfer(from, to, amount), true : false; }
};
```

```
class bank_account {
  int balance_;
  friend void transaction::transfer(bank_account& from, bank_account& to, int amount);
public:
  explicit bank_account(int balance) : balance_{balance} {}
  int balance() const { return balance_; }
};
```

```
error: 'virtual void transaction::transfer(bank_account&, bank_account&, int)' is private within this context
   27 |   friend void transaction::transfer(bank_account& from, bank_account& to, int amount);
      |                                                                          ^
note: declared private here
   13 |   virtual void transfer(bank_account& from, bank_account& to, int amount);
      |                ^~~~~~~~
```

Quiz: Friend Classes

```
class transaction : nonmovable {
  friend class bank_account;
  virtual bool check_balance(const bank_account& from, int amount);
  virtual void transfer(bank_account& from, bank_account& to, int amount);
public:
  virtual ~transaction() {}
  bool run(bank_account& from, bank_account& to, int amount)
  { return check_balance(from, amount) ? transfer(from, to, amount), true : false; }
};
```

```
class bank_account {
  int balance_;
  friend void transaction::transfer(bank_account& from, bank_account& to, int amount);
public:
  explicit bank_account(int balance) : balance_{balance} {}
  int balance() const { return balance_; }
};
```

Quiz: Friend Classes

```
class transaction : nonmovable {
  friend class bank_account;
  virtual bool check_balance(const bank_account& from, int amount);
  virtual void transfer(bank_account& from, bank_account& to, int amount);
public:
  virtual ~transaction() {}
  bool run(bank_account& from, bank_account& to, int amount)
  { return check_balance(from, amount) ? transfer(from, to, amount), true : false; }
};
```

```
class bank_account {
  int balance_;
  friend void transaction::transfer(bank_account& from, bank_account& to, int amount);
public:
  explicit bank_account(int balance) : balance_{balance} {}
  int balance() const { return balance_; }
};
```

```
error: 'bank_account' does not name a type
   12 |   virtual bool check_balance(const bank_account& from, int amount);
      |                                    ^~~~~~~~~~~~
```

Friend Classes

`friend` class declaration does not declare a class.

Friend Classes

`friend` class declaration does not declare a class.

Explicit class declaration is still needed.

Friend Classes

```
class bank_account;

class transaction : nonmovable {
  friend bank_account;
  virtual bool check_balance(const bank_account& from, int amount);
  virtual void transfer(bank_account& from, bank_account& to, int amount);
public:
  virtual ~transaction() {}
  bool run(bank_account& from, bank_account& to, int amount)
  { return check_balance(from, amount) ? transfer(from, to, amount), true : false; }
};
```

```
class bank_account {
  int balance_;
  friend void transaction::transfer(bank_account& from, bank_account& to, int amount);
public:
  explicit bank_account(int balance) : balance_{balance} {}
  int balance() const { return balance_; }
};
```

Friend Classes

```
class bank_account;

class transaction : nonmovable {
  friend bank_account;
  virtual bool check_balance(const bank_account& from, int amount);
  virtual void transfer(bank_account& from, bank_account& to, int amount);
public:
  virtual ~transaction() {}
  bool run(bank_account& from, bank_account& to, int amount)
  { return check_balance(from, amount) ? transfer(from, to, amount), true : false; }
};
```

```
class bank_account {
  int balance_;
  friend void transaction::transfer(bank_account& from, bank_account& to, int amount);
public:
  explicit bank_account(int balance) : balance_{balance} {}
  int balance() const { return balance_; }
};
```

```
bool transaction::check_balance(const bank_account& from, int amount) { return from.balance() >= amount; }
void transaction::transfer(bank_account& from, bank_account& to, int amount) {
  from.balance_ -= amount;
  to.balance_ += amount;
}
```

Friend Non-Member Function Template

```
class bank_account {
  int balance_;
  template<typename Rep>
  friend bool transfer_funds(bank_account& from, bank_account& to, Rep amount);
public:
  explicit bank_account(int balance) : balance_{balance} {}
  int balance() const { return balance_; }
};
```

```
template<typename Rep>
bool transfer_funds(bank_account& from, bank_account& to, Rep amount)
{
  if (from.balance() < amount)
    return false;
  from.balance_ -= amount;
  to.balance_ += amount;
  return true;
}
```

Friend Non-Member Function Template

```
template<typename Rep>
class bank_account;

template<typename Rep>
bool transfer_funds(bank_account<Rep>& from, bank_account<Rep>& to, Rep amount);

template<typename Rep>
class bank_account {
  Rep balance_;
  friend bool transfer_funds<Rep>(bank_account<Rep>& from, bank_account<Rep>& to, Rep amount);
public:
  explicit bank_account(int balance) : balance_{balance} {}
  int balance() const { return balance_; }
};
```

```
template<typename Rep>
bool transfer_funds(bank_account<Rep>& from, bank_account<Rep>& to, Rep amount)
{
  if (from.balance() < amount)
    return false;
  from.balance_ -= amount;
  to.balance_ += amount;
  return true;
}
```

Friend Non-Member Function Template

```
template<typename Rep>
class bank_account;

template<typename Rep>
bool transfer_funds(bank_account<Rep>& from, bank_account<Rep>& to, Rep amount);

template<typename Rep>
class bank_account {
  Rep balance_;
  friend bool transfer_funds<>(bank_account& from, bank_account& to, Rep amount);
public:
  explicit bank_account(int balance) : balance_{balance} {}
  int balance() const { return balance_; }
};
```

```
template<typename Rep>
bool transfer_funds(bank_account<Rep>& from, bank_account<Rep>& to, Rep amount)
{
  if (from.balance() < amount)
    return false;
  from.balance_ -= amount;
  to.balance_ += amount;
  return true;
}
```

Friend Class Template

```
template<typename T>
class A {
  template<typename U>
  friend class B;  // Any specialization of B can access A
};
```

Friend Class Template

```cpp
template<typename T>
class A {
  friend class B<T>;  // Only B<T> can access A<T>
};
```

Friend Class Template

```
template<typename T>
class A {
  friend class B<int>;  // Only B<int> is a friend
};
```

Encapsulation

Encapsulation

> Encapsulation is the art of bundling data and behavior associated with a single responsibility behind a clean interface, concealing implementation details, securing class invariants, and enabling effortless refactoring.
>
> *-- Mateusz Pusz* 😊

Quiz: Can Adding A Member Function Decrease Encapsulation?

Quiz: Can Adding A Member Function Decrease Encapsulation?

If you're writing a function that can be implemented as either a member or as a non-friend non-member, **you should prefer to implement it as a non-member function**. That decision increases class encapsulation. **When you think encapsulation, you should think non-member functions**.

-- Scott Meyers

Quiz: Can Adding A Member Function Decrease Encapsulation?

If you're writing a function that can be implemented as either a member or as a non-friend non-member, **you should prefer to implement it as a non-member function**. That decision increases class encapsulation. **When you think encapsulation, you should think non-member functions**.

-- Scott Meyers

Member functions often increase coupling and decrease encapsulation.

Member Functions May Break Encapsulation

27 Strings library **[strings]**

27.4 String classes **[string.classes]**

27.4.3 Class template basic_string **[basic.string]**

27.4.3.1 General **[basic.string.general]**

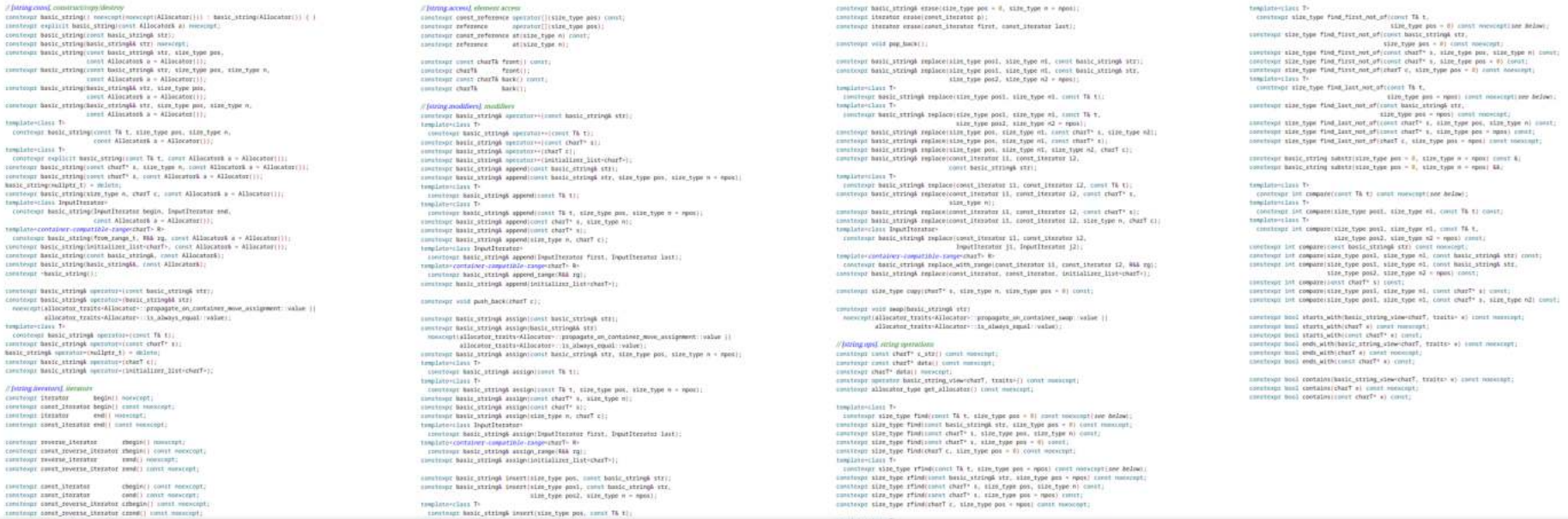
Member Functions May Break Encapsulation

27 Strings library [strings]

27.4 String classes [string.classes]

27.4.3 Class template basic_string [basic.string]

27.4.3.1 General [basic.string.general]



140+ member functions + 25 special member functions

Do Friends Violate Encapsulation?

Do Friends Violate Encapsulation?

A friend function in the class declaration **doesn't violate encapsulation any more than a public member function violates encapsulation**: both have exactly the same authority with respect to accessing the class' non-public parts.

-- C++ FAQ

Friendship In C++ Is Not Inherited

```
class X {
  int value_;
  friend struct Base;
};

struct Base {
  void access(X& x) { ++x.value_; }
};

struct Derived : Base {
  void no_access(X& x) { ++x.value_; }
};
```

Friendship In C++ Is Not Inherited

```cpp
class X {
  int value_;
  friend struct Base;
};

struct Base {
  void access(X& x) { ++x.value_; }
};

struct Derived : Base {
  void no_access(X& x) { ++x.value_; }
};
```

```
In member function 'void Derived::no_access(X&)':
error: 'int X::value_' is private within this context
   62 |   void no_access(X& x) { ++x.value_; }
      |                                ^~~~~
note: declared private here
   53 |   int value_;
      |       ^~~~~
```

Friendship In C++ Is Not Inherited

```cpp
class X {
  int value_;
  friend struct Base;
};

struct Base {
  void access(X& x) { ++x.value_; }
};

struct Derived : Base {
  void no_access(X& x) { ++x.value_; }
};
```

```
In member function 'void Derived::no_access(X&)':
error: 'int X::value_' is private within this context
   62 |   void no_access(X& x) { ++x.value_; }
      |                                ^~~~~
note: declared private here
   53 |   int value_;
      |       ^~~~~
```

Tight coupling is fine for classes that are created and maintained together.
For classes that are created by other users it would cause a maintenance nightmare and prevent any changes to the original type.

Friendship In C++ Is Not Transitive

```
class X {
  int value_;
  friend class Y;
};

class Y {
  int value_;
  friend class Z;
  void access(X& x) { ++x.value_; }
};

class Z {
  void access(Y& y) { ++y.value_; }
  void no_access(X& x) { ++x.value_; }
};
```

Friendship In C++ Is Not Transitive

```cpp
class X {
  int value_;
  friend class Y;
};

class Y {
  int value_;
  friend class Z;
  void access(X& x) { ++x.value_; }
};

class Z {
  void access(Y& y) { ++y.value_; }
  void no_access(X& x) { ++x.value_; }
};
```

```
In member function 'void Z::no_access(X&)':
error: 'int X::value_' is private within this context
   64 |   void no_access(X& x) { ++x.value_; }
      |                               ^~~~~
note: declared private here
   52 |   int value_;
      |       ^~~~~
```

Friendship In C++ Is Not Mutual

```
class Y;

class X {
  int value_;
  friend class Y;
  void no_access(Y& y);
};

class Y {
  int value_;
  void access(X& x) { ++x.value_; }
};

void X::no_access(Y& y) { ++y.value_; }
```

Friendship In C++ Is Not Mutual

```cpp
class Y;

class X {
  int value_;
  friend class Y;
  void no_access(Y& y);
};

class Y {
  int value_;
  void access(X& x) { ++x.value_; }
};

void X::no_access(Y& y) { ++y.value_; }
```

```
In member function 'void X::no_access(Y&)':
error: 'int Y::value_' is private within this context
   64 | void X::no_access(Y& y) { ++y.value_; }
      |                                ^~~~~
note: declared private here
   60 |   int value_;
      |       ^~~~~
```

Rules Of Friendship In C++

Just because I grant you friendship access to me

- doesn't automatically grant your kids access to me,
- doesn't automatically grant your friends access to me,
- and doesn't automatically grant me access to you.

-- C++ FAQ

`friend` Is Not A Unit Testing Solution

Framework.h

```
class Framework {
  int implementation_detail_;
  void more_implementation_details();
  friend class FrameworkTest;
public:
  // ...
};
```

FrameworkTest.cpp

```
#include "Framework.h"

class FrameworkTest {
  // ...
};
```

`friend` Is Not A Unit Testing Solution

Framework.h

```
class Framework {
  int implementation_detail_;
  void more_implementation_details();
  friend class FrameworkTest;
public:
  // ...
};
```

FrameworkTest.cpp

```
#include "Framework.h"

class FrameworkTest {
  // ...
};
```

- **Breaks encapsulation**
  - increases coupling between test and implementation
  - non-breaking changes to private members require modifying tests

`friend` Is Not A Unit Testing Solution

Framework.h

```
class Framework {
  int implementation_detail_;
  void more_implementation_details();
  friend class FrameworkTest;
public:
  // ...
};
```

FrameworkTest.cpp

```
#include "Framework.h"

class FrameworkTest {
  // ...
};
```

- **Breaks encapsulation**
  - increases coupling between test and implementation
  - non-breaking changes to private members require modifying tests
- **Encourages bad design**
  - leads to testing internal details instead of behavior

`friend` Is Not A Unit Testing Solution

1. **Apply Single Responsibility Principle (SRP)**
   - Decompose the monster monolith into smaller testable classes where each has its own responsibility

`friend` Is Not A Unit Testing Solution

1. **Apply Single Responsibility Principle (SRP)**
   - Decompose the monster monolith into smaller testable classes where each has its own responsibility
2. **Use Dependency Injection to improve testability**

`friend` Is Not A Unit Testing Solution

1. **Apply Single Responsibility Principle (SRP)**
   - Decompose the monster monolith into smaller testable classes where each has its own responsibility
2. **Use Dependency Injection to improve testability**
3. **Mock interfaces instead of exposing private details**

`friend` Is Not A Unit Testing Solution

1. **Apply Single Responsibility Principle (SRP)**
   - Decompose the monster monolith into smaller testable classes where each has its own responsibility
2. **Use Dependency Injection to improve testability**
3. **Mock interfaces instead of exposing private details**
4. **Use public getters only when necessary**
   - Do not expose your implementation details to the users unless really needed

Don't Try This At Home!

```
#if BUILD_TESTS
#define private public
#endif
```

Don't Try This At Home!

```
#if BUILD_TESTS
#define private public
#endif
```

```
#if BUILD_TESTS
#define class struct
#endif
```

When `friend` Is Not Needed?

```
template<typename T>
class storage {
  T* buffer_;
public:
  class iterator;
  iterator begin() { return iterator(*this); }
  // ...
};
```

When `friend` Is Not Needed?

```
template<typename T>
class storage {
  T* buffer_;
public:
  class iterator;
  iterator begin() { return iterator(*this); }
  // ...
};
```

```
template<typename T>
class storage<T>::iterator {
  storage* st_;
  // ...
public:
  explicit iterator(storage& st): st_(&st) {}
  iterator& operator++()
  {
    // can access storage private members
    return *this;
  }
};
```

When `friend` Is Not Needed?

```
template<typename T>
class storage {
  T* buffer_;
public:
  class iterator;
  iterator begin() { return iterator(*this); }
  // ...
};
```

```
template<typename T>
class storage<T>::iterator {
  storage* st_;
  // ...
public:
  explicit iterator(storage& st): st_(&st) {}
  iterator& operator++()
  {
    // can access storage private members
    return *this;
  }
};
```

Nested classes have access to the outer class implementation details without the need of **`friend`** keyword usage.

Poor Friends

> Friendship is the strongest coupling we can express in C++, even stronger than inheritance. So we'd better be careful and avoid it if possible.
>
> *-- Arne Mertz*

Poor Friends

> Friendship is the strongest coupling we can express in C++, even stronger than inheritance. So we'd better be careful and avoid it if possible.
>
> *-- Arne Mertz*

- Often seen as an *indicator of poor design*

Poor Friends

> Friendship is the strongest coupling we can express in C++, even stronger than inheritance. So we'd better be careful and avoid it if possible.
>
> *-- Arne Mertz*

- Often seen as an *indicator of poor design*
- Mostly caused by the *lack of friendship granularity*
  - whenever we make a class a **`friend`**, we give it unrestricted access

Poor Friends

> Friendship is the strongest coupling we can express in C++, even stronger than inheritance. So we'd better be careful and avoid it if possible.
>
> *-- Arne Mertz*

- Often seen as an *indicator of poor design*
- Mostly caused by the *lack of friendship granularity*
  - whenever we make a class a **`friend`**, we give it unrestricted access
- *Threatens class' invariants*
  - **`friend`** can mess with our internals as it pleases

Poor Friends

```cpp
class SecureSession {
  friend class SessionFactory;

  // factory needs access
  explicit SecureSession(std::string_view url) noexcept:
    handle_(start_session(url)) {}

  // factory should not have access but has
  static std::string generate_random_token();

  // factory DEFINITELY should not have access but has
  Handle handle_;
  std::string secret_token_ = generate_random_token();
public:
  // ...
};
```

Poor Friends

```cpp
class SecureSession {
  friend class SessionFactory;

  // factory needs access
  explicit SecureSession(std::string_view url) noexcept:
    handle_(start_session(url)) {}

  // factory should not have access but has
  static std::string generate_random_token();

  // factory DEFINITELY should not have access but has
  Handle handle_;
  std::string secret_token_ = generate_random_token();
public:
  // ...
};
```

```cpp
struct SessionFactory {
  std::optional<SecureSession>
    make_secure_session(std::string_view url)
  {
    if (valid_url(url))
      return SecureSession(url);
    return std::nullopt;
  }

  void hack(SecureSession& s)
  {
    s.secret_token_ = "Moo!";  // Yikes!
  }
};
```

Passkey Idiom

```cpp
class SecureSession {
  // private members (no friends anymore)
  Handle handle_;
  std::string secret_token_ = generate_random_token();
  static std::string generate_random_token();

  class ConstructorKey {
    friend class SessionFactory;
    // private members
    ConstructorKey() = default;
    ConstructorKey(const ConstructorKey&) = default;
  };
public:
  // whoever can provide a key has access
  explicit SecureSession(std::string_view url,
                         ConstructorKey) noexcept:
    handle_(start_session(url)) {}
  // ...
};
```

```cpp
struct SessionFactory {
  std::optional<SecureSession>
    make_secure_session(std::string_view url)
  {
    if (valid_url(url))
      return SecureSession(url, {});
    return std::nullopt;
  }

  void hack(SecureSession& s)
  {
    s.secret_token_ = "Moo!";  // Compile-time Error
  }
};
```

Passkey Idiom

```cpp
class SecureSession {
  // private members (no friends anymore)
  Handle handle_;
  std::string secret_token_ = generate_random_token();
  static std::string generate_random_token();

  class ConstructorKey {
    friend class SessionFactory;
    // private members
    ConstructorKey() = default;
    ConstructorKey(const ConstructorKey&) = default;
  };
public:
  // whoever can provide a key has access
  explicit SecureSession(std::string_view url,
                         ConstructorKey) noexcept:
    handle_(start_session(url)) {}
  // ...
};
```

```cpp
struct SessionFactory {
  std::optional<SecureSession>
    make_secure_session(std::string_view url)
  {
    if (valid_url(url))
      return SecureSession(url, {});
    return std::nullopt;
  }

  void hack(SecureSession& s)
  {
    s.secret_token_ = "Moo!";  // Compile-time Error
  }
};
```

A helper type that grants types that can construct it access to selected class member functions.

Passkey Idiom

```
class SecureSession {
  // private members (no friends anymore)
  Handle handle_;
  std::string secret_token_ = generate_random_token();
  static std::string generate_random_token();

  class ConstructorKey {
    friend class SessionFactory;
    // private members
    ConstructorKey() = default;
    ConstructorKey(const ConstructorKey&) = default;
  };
public:
  // whoever can provide a key has access
  explicit SecureSession(std::string_view url,
                         ConstructorKey) noexcept:
    handle_(start_session(url)) {}
  // ...
};
```

- *Before C++20 the default constructor needs to be actually defined*
  - i.e., not defaulted
- Otherwise, it can be created via an *aggregate initialization*

Passkey Idiom

```
class SecureSession {
  // private members (no friends anymore)
  Handle handle_;
  std::string secret_token_ = generate_random_token();
  static std::string generate_random_token();

  class ConstructorKey {
    friend class SessionFactory;
    // private members
    ConstructorKey() = default;
    ConstructorKey(const ConstructorKey&) = default;
  };
public:
  // whoever can provide a key has access
  explicit SecureSession(std::string_view url,
                         ConstructorKey) noexcept:
    handle_(start_session(url)) {}
  // ...
};
```

- *The copy constructor needs to be private*
  - especially if the class is not a private member of **`SecureSession`**
- Otherwise, this hack could give us access too easily

```
ConstructorKey* ptr = nullptr;
SecureSession s("train-it.eu", *ptr);
```

Passkey Idiom

```cpp
class SecureSession {
  // private members (no friends anymore)
  Handle handle_;
  std::string secret_token_ = generate_random_token();
  static std::string generate_random_token();

  class ConstructorKey {
    friend class SessionFactory;
    // private members
    ConstructorKey() = default;
    ConstructorKey(const ConstructorKey&) = default;
  };
public:
  // whoever can provide a key has access
  explicit SecureSession(std::string_view url,
                         ConstructorKey) noexcept:
    handle_(start_session(url)) {}
  // ...
};
```

- *The copy constructor needs to be private*
  - especially if the class is not a private member of **`SecureSession`**
- Otherwise, this hack could give us access too easily

```cpp
ConstructorKey* ptr = nullptr;
SecureSession s("train-it.eu", *ptr);
```

While dereferencing an uninitialized or null pointer is undefined behavior, it will work in all major compilers.

Attorney-Client Idiom

```
class SecureSession {
  // private members (no friends anymore)
  Handle handle_;
  std::string secret_token_ = generate_random_token();
  static std::string generate_random_token();
  explicit SecureSession(std::string_view url) noexcept:
    handle_(start_session(url)) {}
public:
  class FactoryAttorney {
    friend class SessionFactory;
    static SecureSession make(std::string_view url)
    {
      return SecureSession(url);
    }
  };
  // ...
};
```

```
struct SessionFactory {
  std::optional<SecureSession>
    make_secure_session(std::string_view url)
  {
    if (valid_url(url))
      return SecureSession::FactoryAttorney::make(url);
    return std::nullopt;
  }

  void hack(SecureSession& s)
  {
    s.secret_token_ = "Moo!";  // Compile-time Error
  }
};
```

Attorney-Client Idiom

```
class SecureSession {
  // private members (no friends anymore)
  Handle handle_;
  std::string secret_token_ = generate_random_token();
  static std::string generate_random_token();
  explicit SecureSession(std::string_view url) noexcept:
    handle_(start_session(url)) {}
public:
  class FactoryAttorney {
    friend class SessionFactory;
    static SecureSession make(std::string_view url)
    {
      return SecureSession(url);
    }
  };
  // ...
};
```

```
struct SessionFactory {
  std::optional<SecureSession>
    make_secure_session(std::string_view url)
  {
    if (valid_url(url))
      return SecureSession::FactoryAttorney::make(url);
    return std::nullopt;
  }

  void hack(SecureSession& s)
  {
    s.secret_token_ = "Moo!";  // Compile-time Error
  }
};
```

Proxy type that allows a class to expose a part of its private interface to selected types only.

A simple program (RECAP)

```cpp
#include <iostream>

struct my_int {
  int value_;
public:
  constexpr my_int(int value): value_(value) {}
};

int main()
{
  std::cout << my_int{42} << '\n';
}
```

```
error: invalid operands to binary expression ('ostream' (aka 'basic_ostream<char>') and 'my_int')
   11 |   std::cout << my_int{42} << '\n';
      |   ~~~~~~~~~ ^  ~~~~~~~~~~~
```

I Lied 😄

Error message (clang-21.1)

```
<source>:11:13: error: invalid operands to binary expression ('ostream' (aka 'basic_ostream<char>') and 'my_int')
   11 |   std::cout << my_int{42} << '\n';
      |   ~~~~~~~~~ ^  ~~~~~~~~~~
system_error:341:5: note: candidate function template not viable: no known conversion from 'my_int' to 'const error_code' for 2nd argument
  341 |     operator<<(basic_ostream<_CharT, _Traits>& __os, const error_code& __e)
      |     ^                                                ~~~~~~~~~~~~~~~~~~~~~
bits/ostream.h:636:5: note: candidate function template not viable: no known conversion from 'my_int' to 'char' for 2nd argument
  636 |     operator<<(basic_ostream<_CharT, _Traits>& __out, char __c)
      |     ^                                                  ~~~~~~~~
bits/ostream.h:642:5: note: candidate function template not viable: no known conversion from 'my_int' to 'char' for 2nd argument
  642 |     operator<<(basic_ostream<char, _Traits>& __out, char __c)
      |     ^                                               ~~~~~~~~
bits/ostream.h:653:5: note: candidate function template not viable: no known conversion from 'my_int' to 'signed char' for 2nd argument
  653 |     operator<<(basic_ostream<char, _Traits>& __out, signed char __c)
      |     ^                                               ~~~~~~~~~~~~~~
bits/ostream.h:658:5: note: candidate function template not viable: no known conversion from 'my_int' to 'unsigned char' for 2nd argument
  658 |     operator<<(basic_ostream<char, _Traits>& __out, unsigned char __c)
      |     ^                                               ~~~~~~~~~~~~~~~~
bits/ostream.h:667:5: note: candidate function template not viable: no known conversion from 'my_int' to 'wchar_t' for 2nd argument
  667 |     operator<<(basic_ostream<char, _Traits>&, wchar_t) = delete;
      |     ^                                         ~~~~~~~
bits/ostream.h:672:5: note: candidate function template not viable: no known conversion from 'my_int' to 'char8_t' for 2nd argument
  672 |     operator<<(basic_ostream<char, _Traits>&, char8_t) = delete;
      |     ^                                         ~~~~~~~
bits/ostream.h:677:5: note: candidate function template not viable: no known conversion from 'my_int' to 'char16_t' for 2nd argument
  677 |     operator<<(basic_ostream<char, _Traits>&, char16_t) = delete;
      |     ^                                         ~~~~~~~~
bits/ostream.h:681:5: note: candidate function template not viable: no known conversion from 'my_int' to 'char32_t' for 2nd argument
  681 |     operator<<(basic_ostream<char, _Traits>&, char32_t) = delete;
      |     ^                                         ~~~~~~~~
121 lines more...
```

Error message (GCC-15.2)

```
<source>: In function 'int main()':
<source>:11:13: error: no match for 'operator<<' (operand types are 'std::ostream' {aka 'std::basic_ostream<char>'} and 'my_int')
   11 |   std::cout << my_int{42} << '\n';
      |   ~~~~~~~~~ ^~ ~~~~~~~~~~
      |        |       |
      |        |       my_int
      |        std::ostream {aka std::basic_ostream<char>}
<source>:11:13: note: there are 52 candidates
   11 |   std::cout << my_int{42} << '\n';
      |   ~~~~~~~~~~^~~~~~~~~~~~~
In file included from ostream:42,
                 from iostream:43,
                 from <source>:1:
bits/ostream.h:116:7: note: candidate 1: 'std::basic_ostream<_CharT, _Traits>::__ostream_type& std::basic_ostream<_CharT, _Traits>::operator<<(__ostream_type& (*)(__ostream_type&)) [with _CharT = char; _Traits = std::char_traits<char>; __ostream_type = std::b
  116 |       operator<<(__ostream_type& (*__pf)(__ostream_type&))
      |       ^~~~~~~~
bits/ostream.h:116:36: note: no known conversion for argument 1 from 'my_int' to 'std::basic_ostream<char>::__ostream_type& (*)(std::basic_ostream<char>::__ostream_type&)' {aka 'std::basic_ostream<char>& (*)(std::basic_ostream<char>&)'}
  116 |       operator<<(__ostream_type& (*__pf)(__ostream_type&))
      |                  ~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~
bits/ostream.h:125:7: note: candidate 2: 'std::basic_ostream<_CharT, _Traits>::__ostream_type& std::basic_ostream<_CharT, _Traits>::operator<<(__ios_type& (*)(__ios_type&)) [with _CharT = char; _Traits = std::char_traits<char>; __ostream_type = std::basic_ost
  125 |       operator<<(__ios_type& (*__pf)(__ios_type&))
      |       ^~~~~~~~
bits/ostream.h:125:32: note: no known conversion for argument 1 from 'my_int' to 'std::basic_ostream<char>::__ios_type& (*)(std::basic_ostream<char>::__ios_type&)' {aka 'std::basic_ios<char>& (*)(std::basic_ios<char>&)'}
  125 |       operator<<(__ios_type& (*__pf)(__ios_type&))
      |                  ~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~
bits/ostream.h:135:7: note: candidate 3: 'std::basic_ostream<_CharT, _Traits>::__ostream_type& std::basic_ostream<_CharT, _Traits>::operator<<(std::ios_base& (*)(std::ios_base&)) [with _CharT = char; _Traits = std::char_traits<char>; __ostream_type = std::bas
  135 |       operator<<(ios_base& (*__pf) (ios_base&))
      |       ^~~~~~~~
bits/ostream.h:135:30: note: no known conversion for argument 1 from 'my_int' to 'std::ios_base& (*)(std::ios_base&)'
  135 |       operator<<(ios_base& (*__pf) (ios_base&))
      |                  ~~~~~~~~~~~~~^~~~~~~~~~~~~~~~
bits/ostream.h:174:7: note: candidate 4: 'std::basic_ostream<_CharT, _Traits>::__ostream_type& std::basic_ostream<_CharT, _Traits>::operator<<(long int) [with _CharT = char; _Traits = std::char_traits<char>; __ostream_type = std::basic_ostream<char>]'
  174 |       operator<<(long __n)
      |       ^~~~~~~~
bits/ostream.h:174:23: note: no known conversion for argument 1 from 'my_int' to 'long int'
  174 |       operator<<(long __n)
      |                  ~~~~~^~~
bits/ostream.h:178:7: note: candidate 5: 'std::basic_ostream<_CharT, _Traits>::__ostream_type& std::basic_ostream<_CharT, _Traits>::operator<<(long unsigned int) [with _CharT = char; _Traits = std::char_traits<char>; __ostream_type = std::basic_ostream<char>]
  178 |       operator<<(unsigned long __n)
      |       ^~~~~~~~
bits/ostream.h:178:32: note: no known conversion for argument 1 from 'my_int' to 'long unsigned int'
  178 |       operator<<(unsigned long __n)
      |                  ~~~~~~~~~~~~~~^~~
bits/ostream.h:182:7: note: candidate 6: 'std::basic_ostream<_CharT, _Traits>::__ostream_type& std::basic_ostream<_CharT, _Traits>::operator<<(bool) [with _CharT = char; _Traits = std::char_traits<char>; __ostream_type = std::basic_ostream<char>]'
  182 |       operator<<(bool __n)
      |       ^~~~~~~~
bits/ostream.h:182:23: note: no known conversion for argument 1 from 'my_int' to 'bool'
  182 |       operator<<(bool __n)
      |                  ~~~~~^~~
bits/ostream.h:186:7: note: candidate 7: 'std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT, _Traits>::operator<<(short int) [with _CharT = char; _Traits = std::char_traits<char>]'
  186 |       operator<<(short __n);
      |       ^~~~~~~~

327 lines more...
```

Customization Points

Inevitable side effect of popular customization engines (e.g., stream insertion) is a large number of function overloads customizing specific behavior.

Customization Points

Inevitable side effect of popular customization engines (e.g., stream insertion) is a large number of function overloads customizing specific behavior.

Name lookup and overload resolution are often the most expensive parts of compile-time performance in our production projects.

Error Messages For Broken `my_float`

```cpp
class my_float {
  double value_;
public:
  constexpr my_float(double value): value_(value) {}
  // ...
  // no operator+ overloads
};
```

```cpp
std::cout << 1. + my_float{3.14} << '\n';
```

Error Messages For Broken `my_float`

```cpp
class my_float {
  double value_;
public:
  constexpr my_float(double value): value_(value) {}
  // ...
  // no operator+ overloads
};
```

```cpp
std::cout << 1. + my_float{3.14} << '\n';
```

```
error: invalid operands to binary expression ('double' and 'my_float')
   31 |   std::cout << 1. + my_float{3.14} << '\n';
      |                ~~ ^ ~~~~~~~~~~~~~~
note: candidate function not viable: no known conversion from 'my_float' to 'my_int' for 2nd argument
   17 | constexpr my_int operator+(my_int lhs, my_int rhs)
      |                  ^                     ~~~~~~~~~~
1 error generated.
Compiler returned: 1
```

Poll

Have you ever blamed templates for slow compile-times of your projects?

Refactoring `my_int`

```cpp
class my_int {
  int value_;
public:
  constexpr my_int(int value): value_(value) {}

  // Granting access
  friend std::ostream& operator<<(std::ostream& os, my_int mi)
  { return os << mi.value_; }

  friend constexpr my_int operator+(my_int lhs, my_int rhs)
  { return lhs.value_ + rhs.value_; }

  // ...
};
```

Refactoring `my_int`

```cpp
class my_int {
  int value_;
public:
  constexpr my_int(int value): value_(value) {}

  // Granting access
  friend std::ostream& operator<<(std::ostream& os, my_int mi)
  { return os << mi.value_; }

  friend constexpr my_int operator+(my_int lhs, my_int rhs)
  { return lhs.value_ + rhs.value_; }

  // ...
};
```

```cpp
my_int i = 43;
std::cout << i + 1 << '\n';
std::cout << 1 + i << '\n';
```

```
43
43
```

Refactoring `my_int`

```
class my_int {
  int value_;
public:
  constexpr my_int(int value): value_(value) {}

  // Granting access
  friend std::ostream& operator<<(std::ostream& os, my_int mi)
  { return os << mi.value_; }

  friend constexpr my_int operator+(my_int lhs, my_int rhs)
  { return lhs.value_ + rhs.value_; }

  // ...
};
```

- No need to declare anything as everything is defined in a class template
  - no **`friend`** declarations
  - no functions templates forward declarations
- Programming and cognitive cost comparable to a member function

Error Messages For Broken `my_float`

```cpp
class my_float {
  double value_;
public:
  constexpr my_float(double value): value_(value) {}
  // ...
  // no operator+ overloads
};
```

```cpp
std::cout << 1. + my_float{3.14} << '\n';
```

Error Messages For Broken `my_float`

```cpp
class my_float {
  double value_;
public:
  constexpr my_float(double value): value_(value) {}
  // ...
  // no operator+ overloads
};
```

```cpp
std::cout << 1. + my_float{3.14} << '\n';
```

```
error: invalid operands to binary expression ('double' and 'my_float')
   31 | std::cout << 1. + my_float{3.14} << '\n';
      |              ~~ ^ ~~~~~~~~~~~~~
1 error generated.
Compiler returned: 1
```

Error Messages For Broken `my_float`

```cpp
class my_float {
  double value_;
public:
  constexpr my_float(double value): value_(value) {}
  // ...
  // no operator+ overloads
};
```

```cpp
std::cout << 1. + my_float{3.14} << '\n';
```

```
error: invalid operands to binary expression ('double' and 'my_float')
   31 | std::cout << 1. + my_float{3.14} << '\n';
      |              ~~ ^ ~~~~~~~~~~~~~
1 error generated.
Compiler returned: 1
```

`my_int` candidates disappeared from **`my_float`** error message!

Adding abs()

```
class my_int {
public:
  // ...

  friend constexpr my_int abs(my_int);
};

constexpr my_int abs(my_int mi)
{
  return std::abs(mi.value_);
}
```

```
std::cout << abs(my_int{-42}) << '\n';
```

42

Adding abs()

```
class my_int {
public:
  // ...

  friend constexpr my_int abs(my_int);
};

constexpr my_int abs(my_int mi)
{
  return std::abs(mi.value_);
}
```

```
std::cout << abs(my_int{-42}) << '\n';
```

42

```
class my_int {
public:
  // ...

  friend constexpr my_int abs(my_int mi)
  {
    return std::abs(mi.value_);
  }
};
```

```
std::cout << abs(my_int{-42}) << '\n';
```

42

Adding abs()

```
class my_int {
public:
  // ...

  friend constexpr my_int abs(my_int);
};

constexpr my_int abs(my_int mi)
{
  return std::abs(mi.value_);
}
```

```
std::cout << abs(my_int{-42}) << '\n';
std::cout << ::abs(my_int{-42}) << '\n';
```

```
42
42
```

```
class my_int {
public:
  // ...

  friend constexpr my_int abs(my_int mi)
  {
    return std::abs(mi.value_);
  }
};
```

```
std::cout << abs(my_int{-42}) << '\n';
std::cout << ::abs(my_int{-42}) << '\n';
```

```
error: cannot convert 'my_int' to 'int'
   35 | std::cout << ::abs(my_int{-42}) << '\n';
      |                    ^~~~~~~~~~
      |                    |
      |                    my_int
```

Hidden Friends

A **`friend`** function *declared and defined inside* of a class and *taking this class type as a parameter* is called a **Hidden Friend**.

```
struct X {
  friend void func(const X&) { /* ... */ }
};
```

Hidden Friends

A **`friend`** function *declared and defined inside* of a class and *taking this class type as a parameter* is called a **Hidden Friend**.

```
struct X {
  friend void func(const X&) { /* ... */ }
};
```

Such function **can be found only through the ADL**.

Recommendation: Hidden Friends

Prefer Hidden Friend functions rather than global non-member functions **to overload operators or implement other common customization points**. *Do it even when access to the private class members is not required* in the function's definition.

Recommendation: Hidden Friends

Prefer Hidden Friend functions rather than global non-member functions **to overload operators or implement other common customization points**. *Do it even when access to the private class members is not required* in the function's definition.

`friend` functions are not a part of the candidate set for arguments of other types which means they **improve compilation speed** and allow the compiler to present **shorter and clearer compilation errors**.

Summary

`friend`

- **Grants** a function or class **access to all `private` and `protected` members** of another class
- Introduces **strong coupling** between two otherwise independent entities
- Hidden friend function **does not break encapsulation** any more than a member function does
  - no reasons to be afraid of them
- Friendship in C++ is **not**
  - mutual
  - transitive
  - inherited

Summary

USE `friend`

- When **external functions need special access**
  - i.e., functions that can't or shouldn't be implemented as member functions
- When **compilation performance** and **compilation errors clarity** is a concern
  - i.e., operator overloading and other customization points
- When **two classes are tightly coupled but can't be merged together**
  - e.g., have different lifetime

Summary

AVOID `friend`

- When **public interface suffice**
  - exception: hidden friends for customization points
- **`friend`** **should NOT be used for unit testing**
- When **giving access to all implementation details is not desired**
  - consider Passkey or Attorney/Client Idioms to limit the scope of access
- When there are **better alternatives**
  - e.g., dependency injection, SRP

The Art of C++ Friendship

`friend` is a powerful tool, but like art, it requires skill, understanding, and careful application.

The Art of C++ Friendship

`friend` is a powerful tool, but like art, it requires skill, understanding, and careful application.

Avoid overuse, use it judiciously, and appreciate its nuances.

Workshop: C++ Fundamentals You Wish You Had Known Earlier

Congratulations!!!


CAUTION
Programming
is addictive
(and too much fun)