C++ONLINE

MATEUSZ PUSZ

TALK:

THE ART OF C++ FRIENDSHIP

2025

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.

Customization Points

```cpp
#include <iostream>

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

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

Customization Points

```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')
   12 |   std::cout << my_int{42} << '\n';
      |   ~~~~~~~~~ ^  ~~~~~~~~~~~~
```

Customization Points

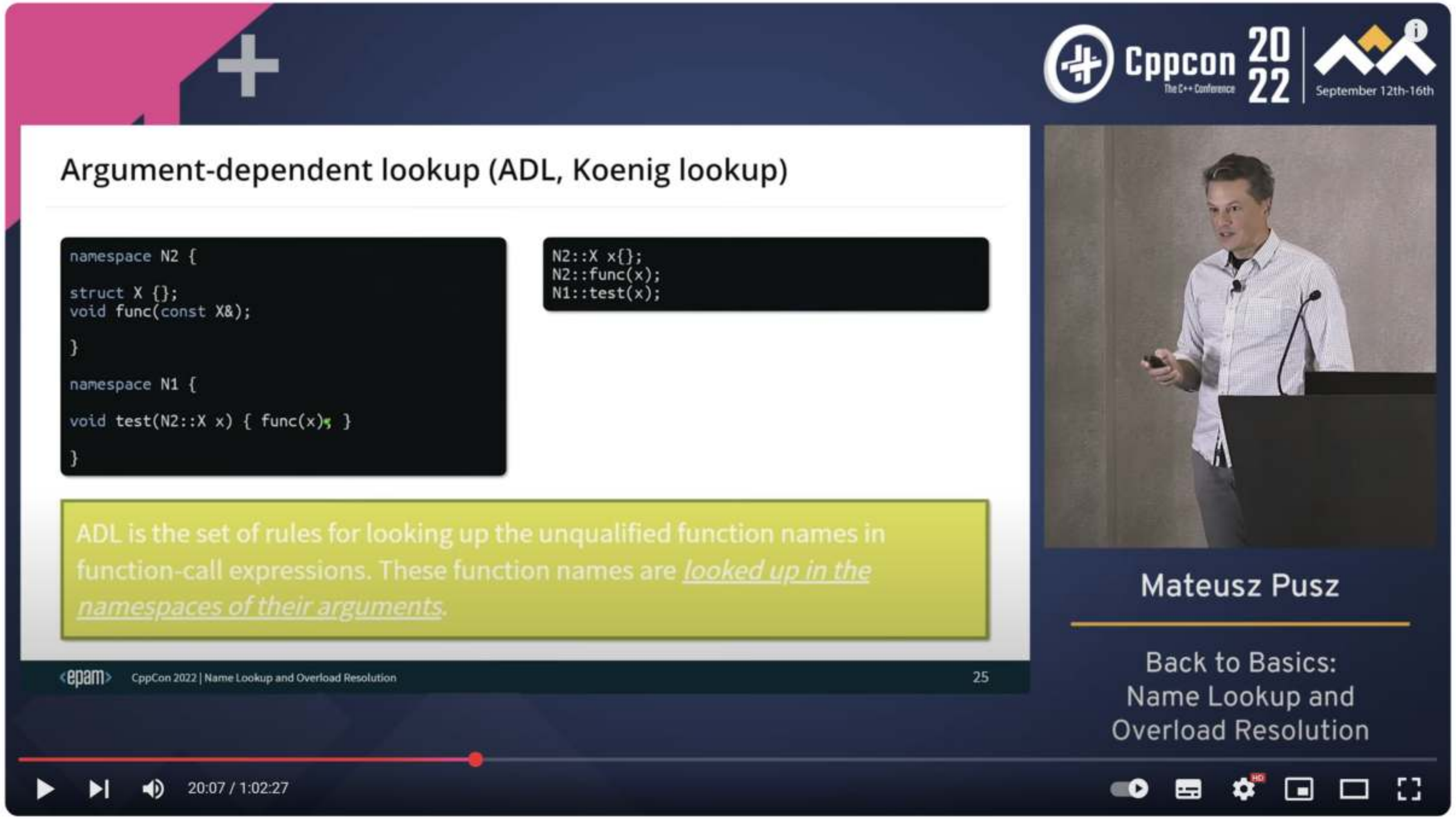
`operator<<` is a customization point that allows fundamental and user defined types to be inserted into the output stream.

Customization Points

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

Most 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

Class Members Access Control

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

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

Class Members Access Control

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

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

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

Class Members Access Control

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

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

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

```
error: 'int my_int::value_' is private within this context
   12 |   return os << si.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

```cpp
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 si)
{
  return os << si.value();
}
```

```cpp
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.

Getters Are Often Not The Solution

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

When done incorrectly can become a safety issue or at least allow the users to break class invariants.

Overloading Binary Operators

```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_;
  }
  // ...
};
```

Overloading Binary Operators

```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
```

Overloading Binary Operators

```
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
```

```
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

```cpp
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();
}
```

```cpp
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 si)
{ return os << si.value_; }

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

Why Does friend Exist?

```
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);
  // ...
};
```

```
std::ostream& operator<<(std::ostream& os, my_int si)
{ return os << si.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';
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 <u>need special access</u> to members.

What is the difference?

```
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);
  // ...
};
```

```
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) {}
  // ...
};
```

What is the difference?

```
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);
  // ...
};
```

```
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) {}
  // ...
};
```

```
my_int i{42};
std::cout << i + 1 << '\n';
std::cout << 1 + i << '\n';
```

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

```
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_; }
};
```

```
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;
}
```

Friend Member Functions

```cpp
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; }
};
```

Friend Member Functions

```cpp
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; }
};
```

```cpp
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 Member Functions

```cpp
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; }
};
```

```cpp
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);
      |                ^~~~~~~~
```

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; }
};
```

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_; }
};
```

Friend Classes

```cpp
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; }
};
```

```cpp
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

```cpp
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_; }
};
```

```cpp
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

```cpp
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_; }
};
```

```cpp
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

```cpp
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_; }
};
```

```cpp
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

```
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 😉

How Non-Member Functions Improve Encapsulation

How Non-Member Functions Improve 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

How Non-Member Functions Improve 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

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

```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_; }
};
```

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_; }
};
```

```
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

```
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

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

FrameworkTest.cpp

```cpp
#include "Framework.h"

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

`friend` Is Not A Unit Testing Solution

Framework.h

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

FrameworkTest.cpp

```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
```

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?

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

```cpp
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 often *breaks encapsulation*

Poor Friends

- Friendship often *breaks encapsulation*
- Often seen as an *indicator of poor design*

Poor Friends

- Friendship often *breaks encapsulation*
- 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 often *breaks encapsulation*
- 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

```
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)) {}
  // ...
};
```

```
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

```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)) {}
  // ...
};
```

- *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

```
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);
```

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

Attorney-Client Idiom

```cpp
class SecureSession {
  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);
    }
  };
  // ...
};
```

```cpp
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

```cpp
class SecureSession {
  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);
    }
  };
  // ...
};
```

```cpp
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.

Why Does `friend` Exist? (RECAP)

```
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);
  // ...
};
```

```
std::ostream& operator<<(std::ostream& os, my_int si)
{ return os << si.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';
std::cout << 1 + i << '\n';
```

43
43

Making `my_int` A Template

```cpp
template<std::integral T>
class my_int;

template<typename T>
std::ostream& operator<<(std::ostream&, my_int<T>);
template<typename T>
constexpr my_int<T> operator+(my_int<T>, my_int<T>);

template<std::integral T>
class my_int {
  T value_;

  // Granting access
  friend std::ostream& operator<< <>(std::ostream&,
                                     my_int);
  friend constexpr my_int operator+ <>(my_int, my_int);
public:
  constexpr my_int(T value): value_(value) {}
  // ...
};
```

```cpp
template<typename T>
std::ostream& operator<<(std::ostream& os, my_int<T> si)
{ return os << si.value_; }

template<typename T>
constexpr my_int<T> operator+(my_int<T> lhs,
                              my_int<T> rhs)
{ return lhs.value_ + rhs.value_; }
```

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

Making `my_int` A Template

```cpp
template<std::integral T>
class my_int;

template<typename T>
std::ostream& operator<<(std::ostream&, my_int<T>);
template<typename T>
constexpr my_int<T> operator+(my_int<T>, my_int<T>);

template<std::integral T>
class my_int {
  T value_;

  // Granting access
  friend std::ostream& operator<< <>(std::ostream&,
                                     my_int);
  friend constexpr my_int operator+ <>(my_int, my_int);
public:
  constexpr my_int(T value): value_(value) {}
  // ...
};
```

```cpp
template<typename T>
std::ostream& operator<<(std::ostream& os, my_int<T> si)
{ return os << si.value_; }

template<typename T>
constexpr my_int<T> operator+(my_int<T> lhs,
                              my_int<T> rhs)
{ return lhs.value_ + rhs.value_; }
```

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

```
error: invalid operands to binary expression ('my_int<int>' and 'int')
   40 | std::cout << i + 1 << '\n';
      |              ~ ^ ~
note: candidate template ignored: could not match 'my_int<T>' against 'int'
   31 | constexpr my_int<T> operator+(my_int<T> lhs, my_int<T> rhs)
      |                     ^
error: invalid operands to binary expression ('int' and 'my_int<int>')
   41 | std::cout << 1 + i << '\n';
      |              ~ ^ ~
note: candidate template ignored: could not match 'my_int<T>' against 'int'
   31 | constexpr my_int<T> operator+(my_int<T> lhs, my_int<T> rhs)
      |                     ^
```

Conversions For (Template) Function Parameters

```
template<typename T>
void foo(T dependent_1, my_int<T> dependent_2, X not_dependent);
```

Conversions For (Template) Function Parameters

```
template<typename T>
void foo(T dependent_1, my_int<T> dependent_2, X not_dependent);
```

DEPENDENT

- Parameter/Argument adjustments
  - e.g., **`std::decay_t`**-like
- Consideration of alternatives
  - e.g., more cv-qualified pointers and references, base classes
- No additional conversions

Conversions For (Template) Function Parameters

```
template<typename T>
void foo(T dependent_1, my_int<T> dependent_2, X not_dependent);
```

DEPENDENT

- Parameter/Argument adjustments
  - e.g., **`std::decay_t`**-like
- Consideration of alternatives
  - e.g., more cv-qualified pointers and references, base classes
- No additional conversions

NOT DEPENDENT

- Trivial Conversions
  - e.g., adding **`const`**
- Promotions
  - e.g., **`short`** -> **`int`**
- Standard Conversions
  - e.g., **`int`** -> **`double`**
- User-defined Conversions
  - i.e., non-explicit constructors and conversion operators

Conversions For (Template) Function Parameters

As conversions are not considered for dependent function parameters, we need to provide more overloads that will handle Interoperability with other convertible types.

Making `my_int` A Template

```
// ...

template<typename T>
constexpr my_int<T> operator+(my_int<T>, T);

template<typename T>
constexpr my_int<T> operator+(T, my_int<T>);

template<std::integral T>
class my_int {
  // ...
  friend constexpr my_int operator+ <>(my_int, T);
  friend constexpr my_int operator+ <>(T, my_int);
public:
  // ...
};
```

```
// ...

template<typename T>
constexpr my_int<T> operator+(my_int<T> lhs, T rhs)
{
  return lhs.value_ + rhs;
}

template<typename T>
constexpr my_int<T> operator+(T lhs, my_int<T> rhs)
{
  return lhs + rhs.value_;
}
```

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

43
43

Making `my_int` A Template

```
// ...

template<std::integral T>
class my_int {
  // ...
public:
  // ...
};
```

```
// ...

template<typename T>
constexpr my_int<T> operator+(my_int<T> lhs, T rhs)
{
  return lhs + my_int{rhs};
}

template<typename T>
constexpr my_int<T> operator+(T lhs, my_int<T> rhs)
{
  return my_int{lhs} + rhs;
}
```

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

43
43

Making `my_int` A Template

```cpp
// ...

template<std::integral T>
class my_int {
  // ...
public:
  // ...
};
```

```cpp
// ...

template<typename T>
constexpr my_int<T> operator+(my_int<T> lhs, T rhs)
{
  return lhs + my_int{rhs};
}

template<typename T>
constexpr my_int<T> operator+(T lhs, my_int<T> rhs)
{
  return my_int{lhs} + rhs;
}
```

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

```
43
43
```

If **T** is cheap to move/copy then we might not need more friends.

Interoperability With Other Representation Types

```
my_int<short> si = my_int{1};
std::cout << my_int{32'767} + si << '\n';
std::cout << si + 32'767 << '\n';
std::cout << short{1} + my_int{32'767} << '\n';
```

Interoperability With Other Representation Types

```cpp
my_int<short> si = my_int{1};
std::cout << my_int{32'767} + si << '\n';
std::cout << si + 32'767 << '\n';
std::cout << short{1} + my_int{32'767} << '\n';
```

```
error: no viable conversion from 'my_int<int>' to 'my_int<short>'
   88 | my_int<short> si = my_int{1};
      |               ^    ~~~~~~~~~
note: candidate constructor (the implicit copy constructor) not viable: no known conversion from 'my_int<int>' to 'const my_int<short> &' for 1st argument
   22 | class my_int {
      |       ^~~~~~
note: candidate constructor (the implicit move constructor) not viable: no known conversion from 'my_int<int>' to 'my_int<short> &&' for 1st argument
   22 | class my_int {
      |       ^~~~~~
note: candidate constructor not viable: no known conversion from 'my_int<int>' to 'short' for 1st argument
   32 |   constexpr my_int(T value): value_(value) {}
      |             ^      ~~~~~~~~
error: invalid operands to binary expression ('my_int<int>' and 'my_int<short>')
   90 | std::cout << my_int{32'767} + si << '\n';
      |              ~~~~~~~~~~~~~~ ^ ~~
note: candidate template ignored: deduced conflicting types for parameter 'T' ('int' vs. 'short')
   47 | constexpr my_int<T> operator+(my_int<T> lhs, my_int<T> rhs)
      |                     ^
note: candidate template ignored: deduced conflicting types for parameter 'T' ('int' vs. 'my_int<short>')
   53 | constexpr my_int<T> operator+(my_int<T> lhs, T rhs)
      |                     ^
note: candidate template ignored: deduced conflicting types for parameter 'T' ('my_int<int>' vs. 'short')
   59 | constexpr my_int<T> operator+(T lhs, my_int<T> rhs)
      |                     ^
error: invalid operands to binary expression ('my_int<short>' and 'int')
   92 | std::cout << si + 32'767 << '\n';
      |              ~~ ^ ~~~~~~
note: candidate template ignored: deduced conflicting types for parameter 'T' ('short' vs. 'int')
   53 | constexpr my_int<T> operator+(my_int<T> lhs, T rhs)
```

Interoperability With Other Representation Types

```cpp
// no need to forward declare operator+ function
// templates anymore

template<std::integral T>
class my_int {
  // ...
  template<std::integral U> friend class my_int;

  template<typename TT, typename U>
  friend constexpr my_int<std::common_type_t<TT, U>>
    operator+(my_int<TT>, my_int<U>);
public:
  template<std::convertible_to<T> U>
  constexpr my_int(my_int<U> other):
    value_(other.value_) {}
  // ...
};
```

```cpp
// ...

template<typename T, typename U>
constexpr my_int<std::common_type_t<T, U>>
  operator+(my_int<T> lhs, my_int<U> rhs)
{
  return lhs.value_ + rhs.value_;
}

template<typename T, typename U>
constexpr my_int<std::common_type_t<T, U>>
  operator+(my_int<T> lhs, U rhs)
{
  return lhs + my_int{rhs};
}

template<typename T, typename U>
constexpr my_int<std::common_type_t<T, U>>
  operator+(T lhs, my_int<U> rhs)
{
  return my_int{lhs} + rhs;
}
```

Interoperability With Other Representation Types

```
my_int<short> si = my_int{1};
std::cout << my_int{32'767} + si << '\n';
std::cout << si + 32'767 << '\n';
std::cout << short{1} + my_int{32'767} << '\n';
```

```
32768
32768
32768
```

Convertibility From `zero`

```
class my_int {
  // ...
};

struct zero {
  constexpr operator my_int() const { return 0; }
};
```

```
my_int i = 42;
std::cout << i + zero{} << '\n';
std::cout << zero{} + i << '\n';
```

42
42

Convertibility From zero

```cpp
class my_int {
  // ...
};

struct zero {
  constexpr operator my_int() const { return 0; }
};
```

```cpp
my_int i = 42;
std::cout << i + zero{} << '\n';
std::cout << zero{} + i << '\n';
```

```
42
42
```

```cpp
template<std::integral T>
class my_int { /* ... */ };

struct zero {
  template<typename T>
  operator my_int<T>() const { return 0; }
};
```

```cpp
my_int i = 42;
std::cout << i + zero{} << '\n';
std::cout << zero{} + i << '\n';
```

```
error: invalid operands to binary expression ('my_int<int>' and 'zero')
   82 | std::cout << i + zero{} << '\n';
      |              ~ ^ ~~~~~~
note: candidate template ignored: could not match 'my_int<U>' against 'zero'
   47 | constexpr my_int<std::common_type_t<T, U>> operator+(my_int<T> lhs, my_int<U> rhs)
      |                                            ^
note: candidate template ignored: substitution failure [with T = int, U = zero]: no type named 'type' in 'std::common_type<int, ze
   53 | constexpr my_int<std::common_type_t<T, U>> operator+(my_int<T> lhs, U rhs)
      |                                            ^
note: candidate template ignored: could not match 'my_int<U>' against 'zero'
   59 | constexpr my_int<std::common_type_t<T, U>> operator+(T lhs, my_int<U> rhs)
      |                                            ^
error: invalid operands to binary expression ('zero' and 'my_int<int>')
   83 | std::cout << zero{} + i << '\n';
      |              ~~~~~~ ^ ~
note: candidate template ignored: could not match 'my_int<T>' against 'zero'
   47 | constexpr my_int<std::common_type_t<T, U>> operator+(my_int<T> lhs, my_int<U> rhs)
      |                                            ^
note: candidate template ignored: could not match 'my_int<T>' against 'zero'
   53 | constexpr my_int<std::common_type_t<T, U>> operator+(my_int<T> lhs, U rhs)
      |                                            ^
note: candidate template ignored: substitution failure [with T = zero, U = int]: no type named 'type' in 'std::common_type<zero, i
   59 | constexpr my_int<std::common_type_t<T, U>> operator+(T lhs, my_int<U> rhs)
      |                                            ^
2 errors generated.
```

Convertibility From zero

```
class my_int {
  // ...
};

struct zero {
  constexpr operator my_int() const { return 0; }
};
```

```
my_int i = 42;
std::cout << i + zero{} << '\n';
std::cout << zero{} + i << '\n';
```

```
42
42
```

```
template<std::integral T>
class my_int { /* ... */ };

struct zero {
  template<typename T>
  operator my_int<T>() const { return 0; }
};
```

```
my_int i = 42;
std::cout << i + zero{} << '\n';
std::cout << zero{} + i << '\n';
```

```
error: invalid operands to binary expression ('my_int<int>' and 'zero')
   82 | std::cout << i + zero{} << '\n';
      |              ~ ^ ~~~~~~
note: candidate template ignored: could not match 'my_int<U>' against 'zero'
   47 | constexpr my_int<std::common_type_t<T, U>> operator+(my_int<T> lhs, my_int<U> rhs)
      |                                             ^
note: candidate template ignored: substitution failure [with T = int, U = zero]: no type named 'type' in 'std::common_type<int, ze
   53 | constexpr my_int<std::common_type_t<T, U>> operator+(my_int<T> lhs, U rhs)
      |                                             ^
note: candidate template ignored: could not match 'my_int<U>' against 'zero'
   59 | constexpr my_int<std::common_type_t<T, U>> operator+(T lhs, my_int<U> rhs)
      |                                             ^
ype<zero, i
```

We could make it work with additional overloads and constraints but most developers would probably give up at this point.

Customization Points (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')
   12 |   std::cout << my_int{42} << '\n';
      |   ~~~~~~~~~ ^  ~~~~~~~~~~
```

I Lied 😁

Error message (clang-19)

```
error: invalid operands to binary expression ('ostream' (aka 'basic_ostream<char>') and 'my_int')
   12 |   std::cout << my_int{42} << '\n';
      |   ~~~~~~~~~ ^  ~~~~~~~~~~~~
basic_ostream.h:530:55: note: candidate function template not viable: no known conversion from 'my_int' to 'char' for 2nd argument
  530 | _LIBCPP_HIDE_FROM_ABI basic_ostream<_CharT, _Traits>& operator<<(basic_ostream<_CharT, _Traits>& __os, char __cn) {
      |                                                       ^                                                 ~~~~~~~~~
basic_ostream.h:557:53: note: candidate function template not viable: no known conversion from 'my_int' to 'char' for 2nd argument
  557 | _LIBCPP_HIDE_FROM_ABI basic_ostream<char, _Traits>& operator<<(basic_ostream<char, _Traits>& __os, char __c) {
      |                                                     ^                                               ~~~~~~~~
basic_ostream.h:562:53: note: candidate function template not viable: no known conversion from 'my_int' to 'signed char' for 2nd argument
  562 | _LIBCPP_HIDE_FROM_ABI basic_ostream<char, _Traits>& operator<<(basic_ostream<char, _Traits>& __os, signed char __c) {
      |                                                     ^                                               ~~~~~~~~~~~~~~~
basic_ostream.h:567:53: note: candidate function template not viable: no known conversion from 'my_int' to 'unsigned char' for 2nd argument
  567 | _LIBCPP_HIDE_FROM_ABI basic_ostream<char, _Traits>& operator<<(basic_ostream<char, _Traits>& __os, unsigned char __c) {
      |                                                     ^                                               ~~~~~~~~~~~~~~~~~
basic_ostream.h:579:1: note: candidate function template not viable: no known conversion from 'my_int' to 'const char *' for 2nd argument
  579 | operator<<(basic_ostream<_CharT, _Traits>& __os, const char* __strn) {
      | ^                                                ~~~~~~~~~~~~~~~~~~
basic_ostream.h:618:53: note: candidate function template not viable: no known conversion from 'my_int' to 'const char *' for 2nd argument
  618 | _LIBCPP_HIDE_FROM_ABI basic_ostream<char, _Traits>& operator<<(basic_ostream<char, _Traits>& __os, const char* __str) {
      |                                                     ^                                               ~~~~~~~~~~~~~~~~~
basic_ostream.h:624:1: note: candidate function template not viable: no known conversion from 'my_int' to 'const signed char *' for 2nd argument
  624 | operator<<(basic_ostream<char, _Traits>& __os, const signed char* __str) {
      | ^                                              ~~~~~~~~~~~~~~~~~~~~~~~~
basic_ostream.h:631:1: note: candidate function template not viable: no known conversion from 'my_int' to 'const unsigned char *' for 2nd argument
  631 | operator<<(basic_ostream<char, _Traits>& __os, const unsigned char* __str) {
      | ^                                              ~~~~~~~~~~~~~~~~~~~~~~~~~~
basic_ostream.h:769:1: note: candidate function template not viable: no known conversion from 'my_int' to 'const error_code' for 2nd argument
  769 | operator<<(basic_ostream<_CharT, _Traits>& __os, const error_code& __ec) {
      | ^                                                ~~~~~~~~~~~~~~~~~~~~~~
128 lines more...
```

Error message (GCC-14)

```
<source>: In function 'int main()':
error: no match for 'operator<<' (operand types are 'std::ostream' {aka 'std::basic_ostream<char>'} and 'my_int')
   12 |   std::cout << my_int{42} << '\n';
      |   ~~~~~~~~~ ^~ ~~~~~~~~~~~~
      |        |       |
      |        |       my_int
      |        std::ostream {aka std::basic_ostream<char>}
In file included from iostream:41,
                 from <source>:1:
ostream:116:7: note: candidate: '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::basic_ostr
  116 |       operator<<(__ostream_type& (*__pf)(__ostream_type&))
      |       ^~~~~~~~
ostream: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&))
      |                  ~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~
ostream:125:7: note: candidate: '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_ostream<char
  125 |       operator<<(__ios_type& (*__pf)(__ios_type&))
      |       ^~~~~~~~
ostream: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&))
      |                  ~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~
ostream:135:7: note: candidate: '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::basic_ostrea
  135 |       operator<<(ios_base& (*__pf) (ios_base&))
      |       ^~~~~~~~
ostream: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&))
      |                  ~~~~~~~~~~~~^~~~~~~~~~~~~~~~~
ostream:174:7: note: candidate: '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)
      |       ^~~~~~~~
ostream:174:23: note:   no known conversion for argument 1 from 'my_int' to 'long int'
  174 |       operator<<(long __n)
      |                  ~~~~~^~~
ostream:178:7: note: candidate: '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)
      |       ^~~~~~~~
ostream:178:32: note:   no known conversion for argument 1 from 'my_int' to 'long unsigned int'
  178 |       operator<<(unsigned long __n)
      |                  ~~~~~~~~~~~~~~^~~
ostream:182:7: note: candidate: '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)
      |       ^~~~~~~~
ostream:182:23: note:   no known conversion for argument 1 from 'my_int' to 'bool'
  182 |       operator<<(bool __n)
      |                  ~~~~~^~~
ostream:186:7: note: candidate: '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);
      |       ^~~~~~~~
ostream:186:24: note:   no known conversion for argument 1 from 'my_int' to 'short int'
  186 |       operator<<(short __n);
      |                  ~~~~~~^~~
ostream:189:7: note: candidate: 'std::basic_ostream<_CharT, _Traits>::__ostream_type& std::basic_ostream<_CharT, _Traits>::operator<<(short unsigned int) [with _CharT = char; _Traits = std::char_traits<char>; __ostream_type = std::basic_ostream<char>]'
  189 |       operator<<(unsigned short __n)
      |       ^~~~~~~~

321 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`

```
template<std::floating_point T>
class my_float {
  T value_;
public:
  constexpr my_float(T value): value_(value) {}
  // ...
  // no operator+ overloads
};
```

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

Error Messages For Broken `my_float`

```cpp
template<std::floating_point T>
class my_float {
  T value_;
public:
  constexpr my_float(T 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<double>')
   84 | std::cout << 1. + my_float{3.14} << '\n';
      |              ~~ ^ ~~~~~~~~~~~~~~
note: candidate template ignored: could not match 'my_int<T>' against 'double'
   47 | constexpr my_int<std::common_type_t<T, U>> operator+(my_int<T> lhs, my_int<U> rhs)
      |                                            ^
note: candidate template ignored: could not match 'my_int<T>' against 'double'
   53 | constexpr my_int<std::common_type_t<T, U>> operator+(my_int<T> lhs, U rhs)
      |                                            ^
note: candidate template ignored: could not match 'my_int' against 'my_float'
   59 | constexpr my_int<std::common_type_t<T, U>> operator+(T lhs, my_int<U> rhs)
      |                                            ^
1 error generated.
Compiler returned: 1
```

Question

Have you ever faced slow compilation due to template overloads?

Refactoring my_int

```
template<std::integral T>
class my_int {
  T value_;
  template<std::integral U> friend class my_int;
public:
  constexpr my_int(T value): value_(value) {}

  template<typename U>
  constexpr my_int(my_int<U> other): value_(other.value_) {}

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

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

  template<typename U>
  friend constexpr my_int<std::common_type_t<T, U>> operator+(my_int lhs, my_int<U> rhs) { return lhs.value_ + rhs.value_; }

  template<typename U>
    requires std::convertible_to<U, my_int<U>>
  friend constexpr my_int<std::common_type_t<T, U>> operator+(my_int lhs, U rhs) { return lhs + my_int<U>{rhs}; }

  template<typename U>
    requires std::convertible_to<U, my_int<U>
  friend constexpr my_int<std::common_type_t<T, U>> operator+(U lhs, my_int rhs) { return my_int<U>{lhs} + rhs; }

  // ...
};
```

Refactoring `my_int`

```
my_int i = 32'767;
std::cout << i + 1 << '\n';
std::cout << 1 + i << '\n\n';

my_int<short> si = i;
std::cout << si + 1 << '\n';
std::cout << 1 + si << '\n\n';

std::cout << i + zero{} << '\n';
std::cout << zero{} + i << '\n\n';

std::cout << i + si << '\n';
```

```
32768
32768


32768
32768

32767
32767

65534
```

Refactoring `my_int`

```
my_int i = 32'767;
std::cout << i + 1 << '\n';
std::cout << 1 + i << '\n\n';

my_int<short> si = i;
std::cout << si + 1 << '\n';
std::cout << 1 + si << '\n\n';

std::cout << i + zero{} << '\n';
std::cout << zero{} + i << '\n\n';

std::cout << i + si << '\n';
```

```
32768
32768

32768
32768

32767
32767

65534
```

- No need to declare anything as everything is defined in a class template
  - no functions templates forward declarations
  - no **`friend`** declarations
- Conversions from **`zero`** work thanks to non-template overload
- The remaining arithmetics with other types work thanks to additional overloads

Error Messages For Broken `my_float`

```
template<std::floating_point T>
class my_float {
  T value_;
public:
  constexpr my_float(T value): value_(value) {}
  // ...
  // no operator+ overloads
};
```

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

Error Messages For Broken `my_float`

```cpp
template<std::floating_point T>
class my_float {
  T value_;
public:
  constexpr my_float(T value): value_(value) {}
  // ...
  // no operator+ overloads
};
```

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

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

Error Messages For Broken `my_float`

```cpp
template<std::floating_point T>
class my_float {
  T value_;
public:
  constexpr my_float(T value): value_(value) {}
  // ...
  // no operator+ overloads
};
```

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

```
error: invalid operands to binary expression ('int' and 'my_float<double>')
   57 | 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()

```
template<std::integral T>
class my_int;

template<typename T>
constexpr my_int<T> abs(my_int<T>);

template<std::integral T>
class my_int {
public:
  // ...

  friend constexpr my_int abs<>(my_int);
};

template<typename T>
constexpr my_int<T> abs(my_int<T> mi)
{ return std::abs(mi.value_); }
```

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

42

Adding abs()

```cpp
template<std::integral T>
class my_int;

template<typename T>
constexpr my_int<T> abs(my_int<T>);

template<std::integral T>
class my_int {
public:
  // ...

  friend constexpr my_int abs<>(my_int);
};

template<typename T>
constexpr my_int<T> abs(my_int<T> mi)
{ return std::abs(mi.value_); }
```

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

42

```cpp
template<std::integral T>
class my_int {
public:
  // ...

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

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

42

Adding abs()

```
template<std::integral T>
class my_int;

template<typename T>
constexpr my_int<T> abs(my_int<T>);

template<std::integral T>
class my_int {
public:
  // ...

  friend constexpr my_int abs<>(my_int);
};

template<typename T>
constexpr my_int<T> abs(my_int<T> mi)
{ return std::abs(mi.value_); }
```

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

```
42
42
```

```
template<std::integral T>
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: no viable conversion from 'my_int<int>' to 'int'
   61 | std::cout << ::abs(my_int{-42}) << '\n';
      |                    ^~~~~~~~~~~
```

Hidden Friends

Friend function *declared and defined inside* of a class and *taking this class type as an argument* is called a **Hidden Friend**.

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

Hidden Friends

Friend function *declared and defined inside* of a class and *taking this class type as an argument* 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**.

Beware Of Asymmetric Conversions

```
template<std::integral T>
class my_int {
public:
  // friend constexpr my_int operator+(my_int lhs, my_int rhs) { return lhs.value_ + rhs.value_; }

  template<typename U>
  friend constexpr my_int<std::common_type_t<T, U>> operator+(my_int lhs, my_int<U> rhs)
  { return lhs.value_ + rhs.value_; }

  // ...
};
```

Beware Of Asymmetric Conversions

```
template<std::integral T>
class my_int {
public:
  // friend constexpr my_int operator+(my_int lhs, my_int rhs) { return lhs.value_ + rhs.value_; }

  template<typename U>
  friend constexpr my_int<std::common_type_t<T, U>> operator+(my_int lhs, my_int<U> rhs)
  { return lhs.value_ + rhs.value_; }

  // ...
};
```

```
my_int i = 32'767;
my_int<short> si = i;
std::cout << zero{} + i << '\n';    // OK
std::cout << i + zero{} << '\n';    // Compile-time Error (no match for 'operator+')
std::cout << i + si << '\n';        // OK
```

Beware Of Asymmetric Conversions

```
template<std::integral T>
class my_int {
public:
  // friend constexpr my_int operator+(my_int lhs, my_int rhs) { return lhs.value_ + rhs.value_; }

  template<typename U>
  friend constexpr my_int<std::common_type_t<T, U>> operator+(my_int lhs, my_int<U> rhs)
  { return lhs.value_ + rhs.value_; }

  template<typename U>
  friend constexpr my_int<std::common_type_t<T, U>> operator+(my_int<U> lhs, my_int rhs)
  { return lhs.value_ + rhs.value_; }

  // ...
};
```

```
my_int i = 32'767;
my_int<short> si = i;
std::cout << zero{} + i << '\n';   // OK
std::cout << i + zero{} << '\n';   // OK
std::cout << i + si << '\n';       // Compile-time Error (ambiguous overload for 'operator+')
```

When Implicit Conversions Are Not Welcomed

```
template<std::integral T>
class my_int {
public:
  // friend constexpr my_int operator+(my_int lhs, my_int rhs) { return lhs.value_ + rhs.value_; }

  template<std::same_as<my_int> Self, typename U>
  friend constexpr my_int<std::common_type_t<T, U>> operator+(Self lhs, my_int<U> rhs)
  { return lhs.value_ + rhs.value_; }

  // ...
};
```

```
my_int i = 32'767;
my_int<short> si = i;
std::cout << zero{} + i << '\n';   // Compile-time Error (no match for 'operator+')
std::cout << i + zero{} << '\n';   // Compile-time Error (no match for 'operator+')
std::cout << i + si << '\n';       // OK
```

When Implicit Conversions Are Not Welcomed

```
template<std::integral T>
class my_int {
public:
  // friend constexpr my_int operator+(my_int lhs, my_int rhs) { return lhs.value_ + rhs.value_; }

  template<std::derived_from<my_int> Self, typename U>
  friend constexpr my_int<std::common_type_t<T, U>> operator+(Self lhs, my_int<U> rhs)
  { return lhs.value_ + rhs.value_; }

  // ...
};
```

```
my_int i = 32'767;
my_int<short> si = i;
std::cout << zero{} + i << '\n';   // Compile-time Error (no match for 'operator+')
std::cout << i + zero{} << '\n';   // Compile-time Error (no match for 'operator+')
std::cout << i + si << '\n';       // OK
```

Hidden Friend Injection Idiom

```
template<std::integral T>
class my_int;

class my_int_base {
  template<typename T, typename U>
  friend constexpr my_int<std::common_type_t<T, U>> operator+(my_int<T> lhs, my_int<U> rhs)
  { return lhs.value_ + rhs.value_; }

  template<typename T, typename U>
    requires std::convertible_to<U, my_int<U>>
  friend constexpr my_int<std::common_type_t<T, U>> operator+(my_int<T> lhs, U rhs)
  { return lhs + my_int{rhs}; }

  template<typename T, typename U>
    requires std::convertible_to<T, my_int<T>>
  friend constexpr my_int<std::common_type_t<T, U>> operator+(T lhs, my_int<U> rhs)
  { return my_int{lhs} + rhs; }

  // ...
};

template<std::integral T>
class my_int : public my_int_base {
  // ..
};
```

Hidden Friend Injection Idiom

Disables implicit (possibly asymmetric) **conversions** for customization point's arguments by *putting all hidden friends as function templates in a base class*.

Hidden Friend Injection Idiom

Disables implicit (possibly asymmetric) **conversions** for customization point's arguments by *putting all hidden friends as function templates in a base class*.

Also, **enables hidden friends usage for a family** of otherwise unrelated types that **satisfy the same concept** while **preserving the most derived class type** for a hidden friend function's logic.

Symbolic Constants With Hidden Friends

```
struct unit_interface;

template<typename T>
concept Unit = SymbolicConstant<T> && std::derived_from<T, unit_interface>;

struct unit_interface {
  template<Unit Lhs, Unit Rhs>
  friend consteval Unit auto operator*(Lhs lhs, Rhs rhs) { /* ... */ }

  template<Unit Lhs, Unit Rhs>
  friend consteval Unit auto operator/(Lhs lhs, Rhs rhs) { /* ... */ }
  // ...
};

template<symbol_text Symbol, QuantityKindSpec auto QS>
struct named_unit : unit_interface { /* ... */ };

template<SymbolicConstant... Expr>
struct derived_unit final : unit_interface { /* ... */ };
```

Symbolic Constants With Hidden Friends

```cpp
struct unit_interface;

template<typename T>
concept Unit = SymbolicConstant<T> && std::derived_from<T, unit_interface>;

struct unit_interface {
  template<Unit Lhs, Unit Rhs>
  friend consteval Unit auto operator*(Lhs lhs, Rhs rhs) { /* ... */ }

  template<Unit Lhs, Unit Rhs>
  friend consteval Unit auto operator/(Lhs lhs, Rhs rhs) { /* ... */ }
  // ...
};

template<symbol_text Symbol, QuantityKindSpec auto QS>
struct named_unit : unit_interface { /* ... */ };

template<SymbolicConstant... Expr>
struct derived_unit final : unit_interface { /* ... */ };
```

```cpp
inline constexpr struct metre final : named_unit<"m", kind_of<isq::length>> {} metre;
inline constexpr struct second final : named_unit<"s", kind_of<isq::time>> {} second;
```

Symbolic Constants With Hidden Friends

```cpp
struct unit_interface;

template<typename T>
concept Unit = SymbolicConstant<T> && std::derived_from<T, unit_interface>;

struct unit_interface {
  template<Unit Lhs, Unit Rhs>
  friend consteval Unit auto operator*(Lhs lhs, Rhs rhs) { /* ... */ }

  template<Unit Lhs, Unit Rhs>
  friend consteval Unit auto operator/(Lhs lhs, Rhs rhs) { /* ... */ }
  // ...
};

template<symbol_text Symbol, QuantityKindSpec auto QS>
struct named_unit : unit_interface { /* ... */ };

template<SymbolicConstant... Expr>
struct derived_unit final : unit_interface { /* ... */ };
```

```cpp
inline constexpr struct metre final : named_unit<"m", kind_of<isq::length>> {} metre;
inline constexpr struct second final : named_unit<"s", kind_of<isq::time>> {} second;
```

```cpp
static_assert(is_of_type<metre / second, derived_unit<metre, per<second>>>);
```

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
- Friendship in C++ is **not**
  - mutual
  - transitive
  - inherited

Summary

USE `friend`

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

Summary

AVOID `friend`

- `friend` should NOT be used for unit testing
- When encapsulation is critical
  - 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.

CAUTION
Programming
is addictive
(and too much fun)