空基类优化（EBCO）

布局原则

Q1 什么是空基类？

C++中的类的定义中，即它们的内部成员变量都在进行声明不占用内存。

- C++还允许类大小为0，比如最坑元素为类时，若类大小为0则数组大小也是0，这会导致很多计算失败，甚然不能存在大小为0的类，但标准规定，空类为基类时，只要不与同一类型的另一个对象或子对象分配在同一地址，就不用分配任何空间

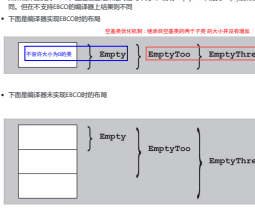
Q2 什么样的类是空类？

典型的空类：只包含(类型或类)非虚成员函数(以及静态数据成员)的类，
因为非静态数据成员、虚函数、以及虚基类，在进行声明时是需要占用内存的。

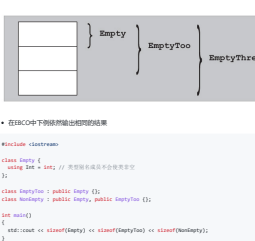
```
#include <iostream>

class Empty {
 using Int = int; // 类型别名或静态变量是安全
};

class EmptyToo : public Empty {};
class EmptyThree : public EmptyToo {};

int main()
{
 std::cout << sizeof(Empty) << sizeof(EmptyToo) << sizeof(EmptyThree);
}
```

- 如果编译器支持EBCO，上述输出结果相同但均不为0，说明EmptyToo中的类Empty没有分配空间，同理类EmptyThree也和类Empty大小相同。但在不支持EBCO的编译器上结果则不同
- 下面是编译器实现EBCO时的布局

空基类优化规则：继承自空基类的两个子类 的大小并没有增加



- 下面是编译器未实现EBCO时的布局



- 在EBCO中下例依然输出相同的结果

```
#include <iostream>

class Empty {
 using Int = int; // 类型别名或静态变量是安全
};

class EmptyToo : public Empty {};
class NonEmpty : public Empty, public EmptyToo {};

int main()
{
 std::cout << sizeof(Empty) << sizeof(EmptyToo) << sizeof(NonEmpty);
}
```

- NonEmpty的基类Empty和EmptyToo不能分配到同一地址空间，否则EmptyToo的基类Empty会和NonEmpty的基类Empty撞在同一地址空间上，即不允许两个相同类型的子对象偏移量相同
- 下面是编译器实现EBCO时NonEmpty的布局

- 对空基类优化规则的根本原因在于，需要能让较两个指针是否指向同一对象，因为指针总是用地址作内部表示，所以必须保证两个不同的指针指向不同的对象，实际应用中许多类都是唯多日一明定义公共typedefs的基类，当这些类作为子对象出现在同一对象中使会出现问题，此时优化后被禁止

成员作为基类

- 对于数据成员则不存在类似EBCO的技术，否则指向成员的指针就会有问题，但模板参数可能是空类

```
template<typename T1, typename T2>
class MyClass {
 private:
  T1 a;
  T2 b;
 ...
};
```

- T1或T2都可能是空类，这样 `MyClass<T1, T2>` 就不能得到最优布局，把模板参数作为基类可以解决这个问题

```
template<typename T1, typename T2>
class MyClass : private T1, private T2 {};
```

- 但T1和T2不一定是类，或者T1和T2是相同类型，如果已知一个模板参数类型为类，另一个类型不是空类，更可行的办法是利用EBCO把可能为空的类型参数与这个成员合并起来

pair

```
template<typename CustomClass>
class Optimizable {
 private:
  CustomClass info; // might be empty
  void* storage;
 ...
};
```

- 改写为

```
template<typename CustomClass>
class Optimizable {
 private:
  BaseMemberPair<CustomClass, void*> info_and_storage;
 ...
};
```

- BaseMemberPair的实现如下，其中的数据成员通过成员函数first()和second()访问

```
template<typename Base, typename Member>
class BaseMemberPair : private Base {
 private:
  Member member;
 public:
  BaseMemberPair(const Base& b, const Member& m)
   : Base(b), member(m) {}

                                  // access base class data via first()
  const Base& first() const       Base const& base() const
  {                               {
   return (const Base&)*this;     return static_cast<Base const&>(*this);
  }                               }

                                  Base& base()
                                  {
                                  return static_cast<Base&>(*this);
                                  }

                                  // access member data via second()
  const Member& second() const    Member const& member() const
  {                               {
   return this->member;             return this->mem;
  }                               }

  Member& second()                Member& member()
  {                               {
   return this->member;             return this->mem;
  }                               }
};
```

奇异递归模板模式（The Curiously Recurring Template Pattern，CRTP）

- CRTP是一种惯用法，形式上是把派生类作为基类的模板参数，最简单的情形如下

```
template<typename T>
class A {};

class B : public A<B> {};
```

- 如果源生类是模板

```
template<typename T>
class A {};

template<typename T>
class B : public A<B<T>> {};
```

- 也可以使用模板的模板参数

```
template<template<typename> class T>
class A {};

template<typename T>
class B : public A<B<T>> {};
```

- CRTP的一个简单应用是记录某个类对象构造的总个数

```
#include <iostream>

template<typename T>
class A {
 private:
  inline static std::size_t i = 0;
 protected:
  A() { ++i; }
  A(const A<T>&) { ++i; }
  A(A<T>&&) { ++i; }
  ~A() { --i; }
 public:
  static std::size_t count() // 返回现存A类型对象的数量
  {
   return i;
  }
};

template<typename T>
class B : public A<B<T>> {};

int main()
{
 std::cout << sizeof(A<int>) << sizeof(A<char>) << sizeof(B<int>);
}
```

- 如果编译器支持EBCO，上述输出结果相同均不为0，说明EmptyToo中的类Empty没有分配空间，同理类EmptyThree也和类Empty大小相同。但在不支持EBCO的编译器上结果则不同
- 下面是编译器实现EBCO时的布局

空基类优化规则：继承自空基类的两个子类 的大小并没有增加

- 下面是编译器未实现EBCO时的布局

- 在EBCO中下例依然输出相同的结果

```
#include <iostream>

class Empty {
 using Int = int; // 类型别名或静态变量是安全
};

class EmptyToo : public Empty {};
class NonEmpty : public Empty, public EmptyToo {};

int main()
{
 std::cout << sizeof(Empty) << sizeof(EmptyToo) << sizeof(NonEmpty);
}
```

- NonEmpty的基类Empty和EmptyToo不能分配到同一地址空间，否则EmptyToo的基类Empty会和NonEmpty的基类Empty撞在同一地址空间上，即不允许两个相同类型的子对象偏移量相同
- 下面是编译器实现EBCO时NonEmpty的布局

- 对空基类优化规则的根本原因在于，需要能让较两个指针是否指向同一对象，因为指针总是用地址作内部表示，所以必须保证两个不同的指针指向不同的对象，实际应用中许多类都是唯多日一明定义公共typedefs的基类，当这些类作为子对象出现在同一对象中使会出现问题，此时优化后被禁止

成员作为基类

- 对于数据成员则不存在类似EBCO的技术，否则指向成员的指针就会有问题，但模板参数可能是空类

```
template<typename T1, typename T2>
class MyClass {
 private:
  T1 a;
  T2 b;
 ...
};
```

- T1或T2都可能是空类，这样 `MyClass<T1, T2>` 就不能得到最优布局，把模板参数作为基类可以解决这个问题

```
template<typename T1, typename T2>
class MyClass : private T1, private T2 {};
```

- 但T1和T2不一定是类，或者T1和T2是相同类型，如果已知一个模板参数类型为类，另一个类型不是空类，更可行的办法是利用EBCO把可能为空的类型参数与这个成员合并起来

pair

```
template<typename CustomClass>
class Optimizable {
 private:
  CustomClass info; // might be empty
  void* storage;
 ...
};
```

- 改写为

```
template<typename CustomClass>
class Optimizable {
 private:
  BaseMemberPair<CustomClass, void*> info_and_storage;
 ...
};
```

- BaseMemberPair的实现如下，其中的数据成员通过成员函数first()和second()访问

```
template<typename Base, typename Member>
class BaseMemberPair : private Base {
 private:
  Member member;
 public:
  BaseMemberPair(const Base& b, const Member& m)
   : Base(b), member(m) {}

                                  // access base class data via first()
  const Base& first() const       Base const& base() const
  {                               {
   return (const Base&)*this;     return static_cast<Base const&>(*this);
  }                               }

                                  Base& base()
                                  {
                                  return static_cast<Base&>(*this);
                                  }

                                  // access member data via second()
  const Member& second() const    Member const& member() const
  {                               {
   return this->member;             return this->mem;
  }                               }

  Member& second()                Member& member()
  {                               {
   return this->member;             return this->mem;
  }                               }
};
```

奇异递归模板模式（The Curiously Recurring Template Pattern，CRTP）

- CRTP是一种惯用法，形式上是把派生类作为基类的模板参数，最简单的情形如下

```
template<typename T>
class A {};

class B : public A<B> {};
```

- 如果源生类是模板

```
template<typename T>
class A {};

template<typename T>
class B : public A<B<T>> {};
```

- 也可以使用模板的模板参数

```
template<template<typename> class T>
class A {};

template<typename T>
class B : public A<B<T>> {};
```

- CRTP的一个简单应用是记录某个类对象构造的总个数

```
#include <iostream>

template<typename T>
class A {
 private:
  inline static std::size_t i = 0;
 protected:
  A() { ++i; }
  A(const A<T>&) { ++i; }
  A(A<T>&&) { ++i; }
  ~A() { --i; }
 public:
  static std::size_t count() // 返回现存A类型对象的数量
  {
   return i;
  }
};

template<typename T>
class B : public A<B<T>> {};

int main()
{
 std::cout << sizeof(A<int>) << sizeof(A<char>) << sizeof(B<int>);
}
```

- 这里结合了CRTP与Barton-Nackman trick，基类使用CRTP为派生类提供一个基于派生类 `operator==` 定义的 `operator!=` ，使用Barton-Nackman trick通过一个友元函数提供了定义。结合Barton-Nackman trick的CRTP可以为许多运算符提供基于一些权威运算符的通用定义，这使其成为了C++模板库作者最喜爱的技术

外观模式（Facade）

- 结合Barton-Nackman trick与CRTP定义迭代器并是一个很方便的捷径，把这个惯用扩展开，CRTP基类可以根据由CRTP派生类暴露的少得多的接口定义大部分public接口，这个模式称为facade模式，这个模式在定义需要满足一些现有接口（数字类型、迭代器、容器等）的新类型时十分有用。为了阐述facade模式，首先实现一个迭代器facade，它能收入现代一个普通标准库要求的迭代器的实现。迭代器类型（尤其是高级和认识迭代器，以此类推，projection函数获得提交投影迭代器十分简单
- 其中最略的对IteratorFacade的迭代器的部分实现如下

```
template<typename Derived, typename Value, typename Category,
 typename Reference = Value&, typename Distance = std::ptrdiff_t>
class IteratorFacade {
 public:
  using value_type = std::remove_const_t<Value>;
  using reference = Reference;
  using pointer = Value*;
  using difference_type = Distance;
  using iterator_category = Category;

  // input iterator interface:
  reference operator*() const { ... }
  pointer operator->() const { ... }
  Derived& operator++() { ... }
  Derived operator++(int) { ... }
  friend bool operator==(const IteratorFacade& lhs, const IteratorFacade& rhs)
   const IteratorFacade& rhs) { ... }
  ...

  // bidirectional iterator interface:
  Derived& operator--() { ... }
  Derived operator--(int) { ... }

  // random access iterator interface:
  reference operator[](difference_type n) const { ... }
  Derived& operator+=(difference_type n) { ... }
  ...
  friend difference_type operator-(const IteratorFacade& lhs,
   const IteratorFacade& rhs) { ... }
  friend bool operator<(const IteratorFacade& lhs,
   const IteratorFacade& rhs) { ... }
 ...
};
```

- 为了简洁省略了一些声明，然而实现列出的这些也足够麻烦。幸运的是，这些接口能提取一些核心操作
  - 对所有迭代器
  - dereference()：访问迭代器值（常通过使用operator*和->）
  - increment()：移动迭代器指向序列中的下个项目
  - equals()：比较两个迭代器是否指向同一个项目
  - 对双向迭代器
  - decrement()：移动迭代器指向序列表中的前一个项目
  - 对随机访问迭代器
  - advance()：往前或往后移动迭代器n步
  - measureDistance()：计算两个迭代器之间的距离
- facade的角色是适配一个可实现核心操作接供的类型，从而提供给所有迭代器接口。IteratorFacade的实现主要涉及将迭代器语句映射到最小接口，提供一个成员函数asDerived()访问CRTP派生类

```
Derived& asDerived()
{
 return *static_cast<Derived*>(this);
}

const Derived& asDerived() const
{
 return *static_cast<const Derived*>(this);
}
```

- 通过这个定义就能直接实现大部分facade接口

```
reference operator*() const
{
 return asDerived().dereference();
}

Derived& operator++()
{
 asDerived().increment();
 return asDerived();
}

Derived operator++(int)
{
 Derived result(asDerived());
 asDerived().increment();
 return result;
}

friend bool operator==(const IteratorFacade& lhs, const IteratorFacade& rhs)
{
 return lhs.asDerived().equals(rhs.asDerived());
}

...
```

- 使用IteratorFacade可以很轻松地定义一个链表类的迭代器

```
template<typename T>
class ListNode {
 public:
  T value;
  ListNode(T x) : value(x) {}
  ListNode<T>* next = nullptr;
  ~ListNode() { delete next; }
};

template<typename T>
class ListNodeIterator
: public IteratorFacade<ListNodeIterator<T>, T, std::forward_iterator_tag> {
 public:
  T& dereference() const
  {
   return current->value;
  }
  void increment()
  {
   current = current->next;
  }
  bool equals(const ListNodeIterator& other) const
  {
   return current == other.current;
  }
  ListNodeIterator(ListNode<T>* current = nullptr) : current(current) {}
 private:
  ListNode<T>* current = nullptr;
};
```

- 这个迭代器实现上的一个缺点是，需要将操作暴露为public接口，为了隐藏接口，可以修改IteratorFacade，使其通过一个分离的访问类IteratorFacadeAccess在CRTP派生类上执行操作

```
class IteratorFacadeAccess {
 template<typename Derived, typename Value, typename Category,
   typename Reference, typename Distance>
 friend class IteratorFacade;

 // 所有迭代器的要求
 template<typename Reference, typename Iterator>
 static Reference dereference(const Iterator& i)
 {
  return i.dereference();
 }
 ...

 // 双向迭代器的要求
 template<typename Iterator>
 static void decrement(Iterator& i)
 {
  return i.decrement();
 }

 // 随机访问迭代器的要求
 template<typename Iterator, typename Distance>
 static void advance(Iterator& i, Distance n)
 {
  return i.advance(n);
 }
 ...
};
```

- 迭代器将这个类指定为友元即可将IteratorFacade需要的接口设为private

```
friend class IteratorFacadeAccess;
```

- IteratorFacade使得建立一个迭代器适配器很简单。比如有一个Person容器

```
struct Person {
 std::string firstName;
 std::string lastName;
 friend std::ostream& operator<<(std::ostream& os, const Person& p)
 {
  return os << p.lastName << ", " << p.firstName;
 }
};
```

- 现在只想看到firstName而不需要所有数据，这需要开发一个ProjectionIterator迭代器适配器，它允许将基本迭代器的值投影到某个数据成员的指针上，如 `Person::firstName`

```
template<typename Iterator, typename T>
class ProjectionIterator
: public IteratorFacade<
 ProjectionIterator<Iterator, T>,
 T,
 typename std::iterator_traits<Iterator>::iterator_category,
 T&,
 typename std::iterator_traits<Iterator>::difference_type>
{
 using Base = typename std::iterator_traits<Iterator>::value_type;
 using Distance = typename std::iterator_traits<Iterator>::difference_type;

 Iterator iter;
 T Base::* member;

 friend class IteratorFacadeAccess;
 ... // implement core iterator operations for IteratorFacade
public:
 ProjectionIterator(Iterator iter, T Base::* member)
 : iter(iter), member(member) {}
};

template<typename Iterator, typename Base, typename T>
auto project(Iterator iter, T Base::* member)
{
 return ProjectionIterator<Iterator, T>(iter, member);
}
```

- 每个投影迭代器保存两个值，iter和member，iter是基本序列的迭代器，member是一个数据成员的指针。IteratorFacade的模板参数中，第一个是ProjectionIterator本身（以启用CRTP），第二个（T）和第四个（T&）是投影迭代器的值和引用类型，第三个和第五个只是通过基本迭代器的类型和不同类型标准，因此当iterator是一个输入迭代器时，投影迭代器也是输入迭代器。iterator是双向迭代器则投影迭代器也是双向迭代器，以此类推。project()函数获得建立投影迭代器十分简单
- 其中略略的对IteratorFacade的迭代器的部分实现如下

```
T& dereference() const
{
 return (*iter).*member;
}

void increment()
{
 ++iter;
}

bool equals(const ProjectionIterator& other) const
{
 return iter == other.iter;
}

void decrement()
{
 --iter;
}
```

- 使用投影迭代器，可以打印一个包含Person值的vector中的firstName

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

int main()
{
 std::vector<Person> authors =
   { {"David", "Vandevoorde"},
   {"Nicolai", "Josuttis"},
   {"Douglas", "Gregor"} };
 std::copy(project(authors.begin(), &Person::firstName),
   project(authors.end(), &Person::firstName),
   std::ostream_iterator<std::string>{std::cout, "\n"});
}

// output
David
Nicolai
Douglas
```

Mixins

- 考虑一个由一系列点组成的Polygon类

```
class Point {
 public:
  double x, y;
  Point() : x(0.0), y(0.0) {}
  Point(double x, double y) : x(x), y(y) {}
};

class Polygon {
 private:
  std::vector<Point> points;
 public:
  ... // public operations
};
```

- 如果用户可以扩展每个点关联的信息来包含特定应用的数据，如颜色，这个类将会更有用，因此参数化点的类型

```
template<typename T>
class Polygon {
 private:
  std::vector<T> points;
 public:
  ... // public operations
};
```

- 用户可以使用继承来创建自定义类型的点，但这要求Point类型暴露给用户来继承，并且要小心提供和Point完全一样的接口，否则不能用于Polygon

```
class LabeledPoint : public Point {
 public:
  std::string label;
  LabeledPoint() : Point(), label("") {}
  LabeledPoint(double x, double y) : Point(x, y), label("") {}
};
```

- 使用Mixins手法即可在不使用继承的情况下定制行为，由此可以很方便地引入额外信息而不改变接口

```
template<typename... Mixins>
class Point : public Mixins... {
 public:
  double x, y;
  Point() : Mixins()..., x(0.0), y(0.0) {}
  Point(double x, double y) : Mixins()..., x(x), y(y) {}
};

class Label {
 public:
  std::string label;
  Label() : label("") {}
};

using LabeledPoint = Point<Label>;

// 可以加入多个服务
class Color {
 public:
  unsigned char red = 0, green = 0, blue = 0;
};

using MyPoint = Point<Label, Color>;
```

- 把mixins提供给Polygon类模板本身即可完全隐藏Point类

```
template<typename... Mixins>
class Polygon {
 private:
  std::vector<Point<Mixins...>> points;
 public:
  ... // public operations
};
```

- mixin结合CRTP将变得更强大，下面是一个CRTP-mixin版本的Point，每个mixins是一个类模板，于是现在被加入的属性必须是类模板，即Label和Color得类需要定为类模板。现在可以把之前的计数模板加入Point来计算由Polygon创建的点数量

```
template<template<typename>class... Mixins>
class Point : public Mixins<Point>>... {
 public:
  double x, y;
  Point() : Mixins<Point>>()..., x(0.0), y(0.0) {}
  Point(double x, double y) : Mixins<Point>>()..., x(x), y(y) {}
};

template<typename T>
class A {
 private:
  inline static std::size_t i = 0;
 protected:
  A() { ++i; }
  A(const A<T>&) { ++i; }
  A(A<T>&&) { ++i; }
  ~A() { --i; }
 public:
  static std::size_t count()
  { // 返回现存A类型对象的数量
   return i;
  }
};

using PointCount = Point<A>;

int main()
{
 PointCount a, b, c;
 std::cout << a.count(); // 3
 std::cout << PointCount::count(); // 3
}
```

- Mixins可以参数化成员函数的虚拟性（virtuality，即是否为虚函数）

```
#include <iostream>

class A {};

class B {
 public:
  virtual void f() {}
};

template<typename... Mixins>
class Base : private Mixins... {
 public:
  // 有的因虚函数覆盖而被子Mixins中f()的声明
  void f() { std::cout << "Base::f()" << "\n"; }
};

template<typename... Mixins>
class Derived : public Base<Mixins...> {
 public:
  void f() { std::cout << "Derived::f()" << "\n"; }
};

int main()
{
 Base<A>* p = new Derived<A>;
 p->f(); // calls Base::f()

 Base<B>* q = new Derived<B>;
 q->f(); // calls Derived::f()
}
```

命名的模板实参（Named Template Argument）

- 模板常常带有一长串类型参数，不过通常都设有默认值

```
class A {};
class B {};
class C {
 public:
  static void print()
  {
   std::cout << 1;
  }
};
class D {};

template<typename T1 = A, typename T2 = B, typename T3 = C, typename T4 = D>
class MyClass {};
```

- 现在想指定某个实参，而其他参数依然使用默认实参，比如将T3指定为E

```
MyClass<A, B, E> a;
```

- 但这样还需要将指定位置之前的默认实参也写出来，十分繁琐，此时使希望只指定一次就达到相同效果

```
MyClass<SetT3<E>> a;
```

- 要达到这个目的需要将原来的默认实参的别名统一到一个类中，通过指定别名即可使用所需的默认实参

```
class Alias {
 public:
  using P1 = A;
  using P2 = B;
  using P3 = C;
  using P4 = D;
};

template<typename T1 = Alias, typename T2 = Alias,
 typename T3 = Alias, typename T4 = Alias>
class MyClass {
 private:
  void f() { T1::P3::print(); }
};

MyClass a; // 使用默认实参
a.f(); // 1: 调用C::print()

template<typename T>
class SetT1 : public Alias {
 public:
  using P3 = T;
};

class E {
 public:
  static void print()
  {
   std::cout << 2;
  }
};

MyClass<SetT1<E>> b;
b.f(); // 2: 调用E::print()
```

- 上述只重写了P3，根据需要再把其他情况补充完整即可

```
template<typename T>
class SetT1 : public Alias {
 public:
  using P1 = T;
};

template<typename T>
class SetT2 : public Alias {
 public:
  using P2 = T;
};

template<typename T>
class SetT3 : public Alias {
 public:
  using P3 = T;
};

template<typename T>
class SetT4 : public Alias {
 public:
  using P4 = T;
};
```

- 为了方便，再把参数统一到一个派生类（因为还可能有其他不需要这样处理的参数）

```
template<typename T1 = Alias, typename T2 = Alias,
 typename T3 = Alias, typename T4 = Alias>
class MyClass {
 using Policies = SetBase<T1, T2, T3, T4>; // 指定T1. T2. T3. T4为基类
 public:
  void f() { Policies::P3::print(); }
};
```

- 但不能直接从多个相同的类直接继承

```
template<typename T1, typename T2>
setBase : public T1, public T2 {};

setBase<Alias, Alias> x; // 错误，重复指定基类
```

- 因此需要一个用于生成不同类的中间层

```
template<typename T, int N>
class Mid : public T {};

template<typename T1, typename T2, typename T3, typename T4>
class SetBase : // 把T1. T2. T3. T4的为基类
 public Mid<T1, 1>,
 public Mid<T2, 2>,
 public Mid<T3, 3>,
 public Mid<T4, 4>
{};
```

- 但中间层还是从单个原始类派生的，为了防止多次继承产生二义性，不能直接将别名类用作基类，而是从别名类虚派生一个类用作基类

```
class Args : virtual public Alias {}; // Args包含了现在P1. P2. P3. P4
```

- 现在应当使用这个虚派生的类作为新的默认实参，此外，用于重写的类也应该使用虚继承。最终所有实现如下

```
#include <iostream>

// 要使用的默认实参
class A {};
class B {};
class C {
 public:
  static void print()
  {
   std::cout << 1;
  }
};
class D {};

// 包含默认实参别名的类，它的作为新的默认实参
// 要使用别名的默认实参：Base 明示需要指定Alias::P1
class Alias {
 public:
  using P1 = A;
  using P2 = B;
  using P3 = C;
  using P4 = D;
};

// 覆盖别名的类，用来指定某个位置的参数
// 比如MyClass<SetT3<E>>就是将MyClass的模板参数设置为SetT3<E>
// SetT3<E>中只改写了P3 = E，其他别名依然与上述类相同
template<typename T>
class SetT1 : virtual public Alias { // 使用虚继承
 public:
  using P1 = T;
};

template<typename T>
class SetT2 : virtual public Alias {
 public:
  using P2 = T;
};

template<typename T>
class SetT3 : virtual public Alias {
 public:
  using P3 = T;
};

template<typename T>
class SetT4 : virtual public Alias {
 public:
  using P4 = T;
};

// 由于不能从多个相同类直接继承，需要一个中间层用于派分
template<typename T, int N>
class Mid : public T {};

template<typename T1, typename T2, typename T3, typename T4>
class SetBase : // 把T1. T2. T3. T4的为基类
 public Mid<T1, 1>,
 public Mid<T2, 2>,
 public Mid<T3, 3>,
 public Mid<T4, 4>
{};

// Alias要被用作默认实参，但SetBase会将其多次指定为Mid的基类
// 为了防止多次继承产生二义性，创新生一个新类作代Alias作为默认实参
class Args : virtual public Alias {}; // Args即包含了现在P1. P2. P3. P4

// 最终类的默认实参都使用被间接的Args
template<typename T1 = Args, typename T2 = Args,
 typename T3 = Args, typename T4 = Args>
class MyClass {
 using Policies = SetBase<T1, T2, T3, T4>; // 即把Args作为基类
 public:
  void f()
  {
   Policies::P3::print(); // P3的C. 因此就是调用C::print()
  }
};

// 指定之一个类，用作下面的指定参数
class E {
 public:
  static void print()
  {
   std::cout << 2;
  }
};

int main()
{
 MyClass a; // 使用默认实参
 a.f(); // 1: 调用C::print()
 MyClass<SetT3<E>> b; // P3指定是E的别名，相当于指定T3为E
 b.f(); // 2: 调用E::print()
}
```