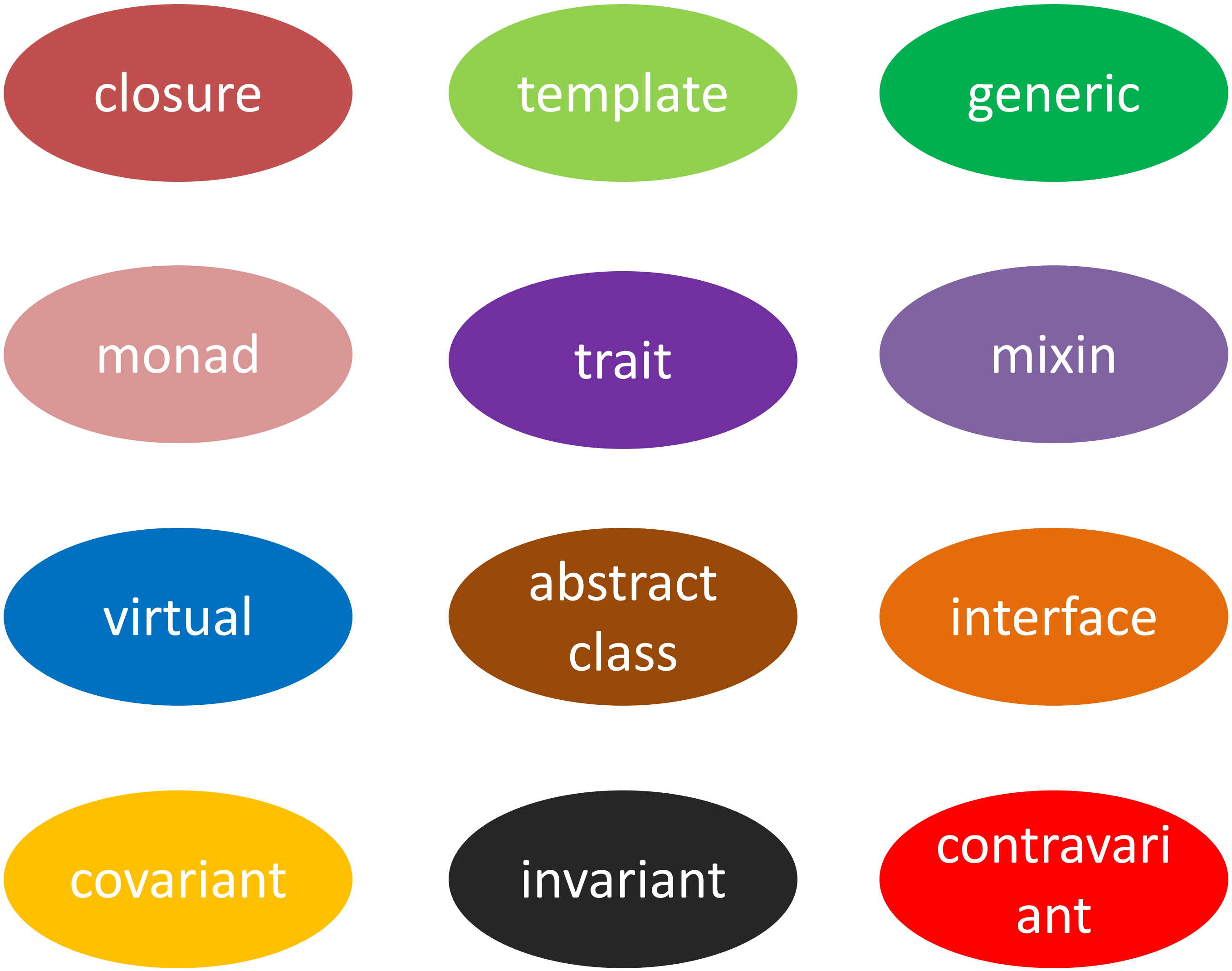
Lecture 11

语言功能和设计模式

徐 辉
xuh@fudan.edu.cn


closure
template
generic
monad
trait
mixin
virtual
abstract class
interface
covariant
invariant
contravariant

大纲

- 一、常用功能
- 二、智能指针
- 三、代码复用和继承
- 四、子类型和协变
- 五、函数式编程

一、常用功能

TeaPL实现自举还需要哪些功能？

结构体函数成员变量

C语言风格

```
struct Point {
    int x;
    int y;
    double (*len)(struct Point *self);//函数指针
}
int getl(struct Point *self) { return x+y; }
struct Point point = {1, 1};
point.len = getlen;
double distance = point.len(&point1);
```

C++风格

```
struct Point {  //class A
    int x;
    int y;
    Point(int x, int y) : x(x), y(y) {}
    int len(){ return x+y; } //成员函数
}
//double Point::len() { return sqrt(x*x + y*y);  }
```

Rust风格

```
struct Point { x:i32, y:i32; }
impl Point {
    fn len(&self) -> i32 { //成员函数
        self.x + self.y
    }
}
```

如何实现成员函数

```
clang++ -S -emit-llvm $1
```

```
struct Point {  //class A
    int x;
    int y;
    int len(){
        return x+y;
    }
};

int main(){
    Point x = {1,1};
    x.len();
}
```

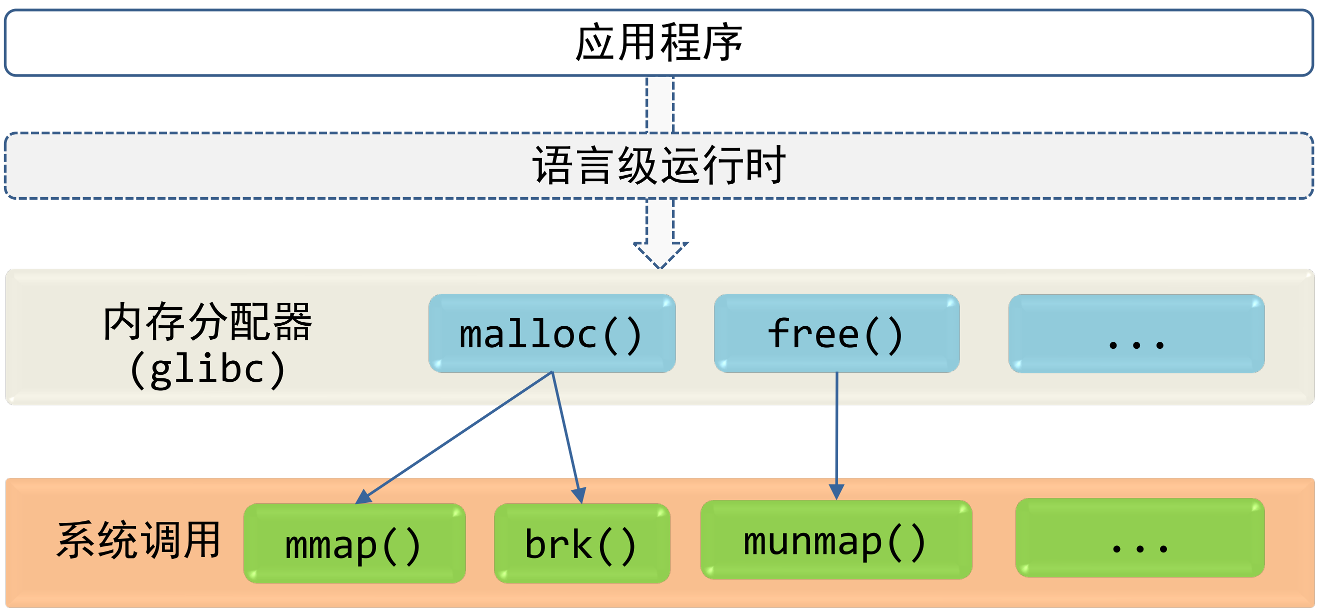
```
%Point = type { i32, i32 }


define int @A.len(Point* %0) {
    ...
}
```

构造函数？

写出完整的线性IR

如何实现动态内存管理？

- 应用：动态数组、链表
- 基于glibc里的dlmalloc或ptmalloc：
  - 分配：malloc(size_t n)
    - 分配n个字节的空间，并返回指向该内存的指针
  - 释放：free(void * p)

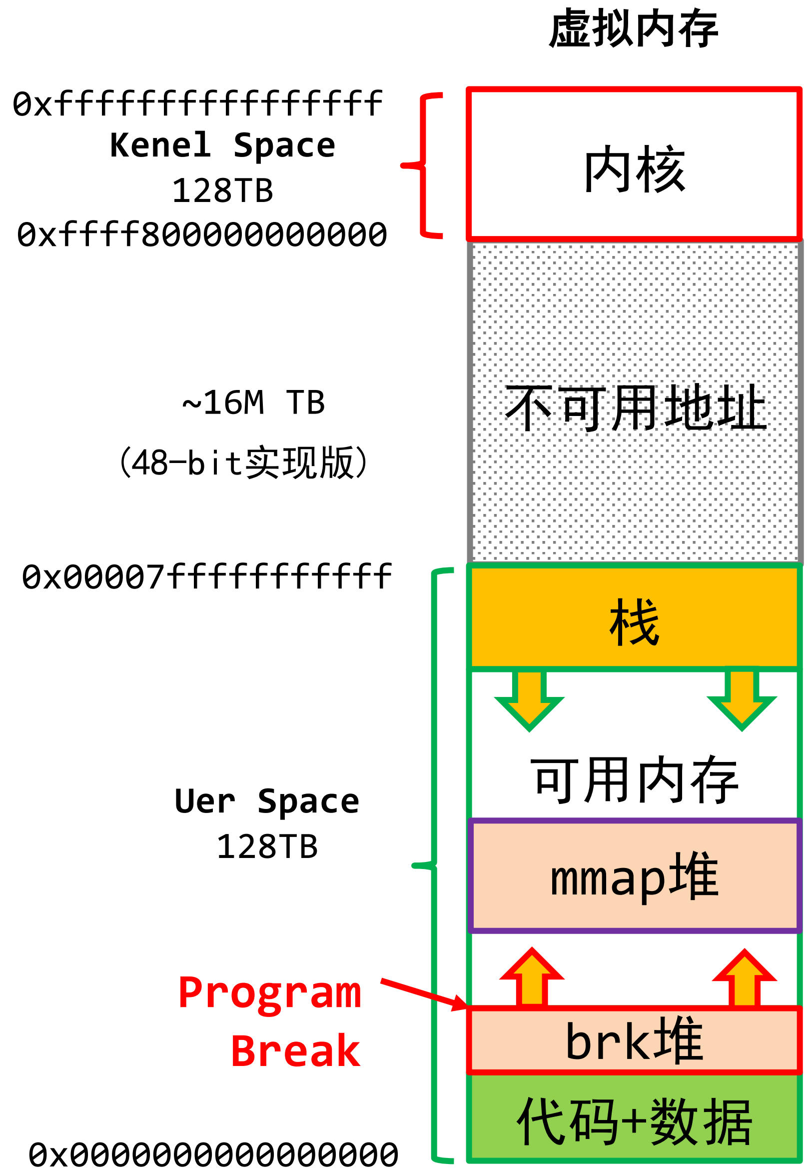


调用外部库函数（libc）

```
@str = private unnamed_addr constant [8 x i8] c"Hi:%d!\0A\00"

declare ptr @malloc(i64 noundef)
declare void @free(ptr noundef)
declare i32 @printf(i8* noundef, ...)

define void @main( ) {
bb0:
   %r100 = call ptr @malloc(i64 1024)
   call void @free(ptr %r100)
   %r101 = getelementptr [8 x i8], [8 x i8]* @str, i64 0, i64 0
   %r102 = call i32 (i8*, ...) @printf(i8* %r101, i32 100)
   ret void
}
```

内存管理

- 栈：编译时确定开销
  - 新的函数调用会创建栈帧
  - 调用规约
  - 函数返回自动退栈
- 堆：动态管理
  - 用户态：系统调用（如dlmalloc）
    - 小于阈值时使用：brk
    - 大于阈值时：mmap
  - 内核态：Buddy Allocator/Slab



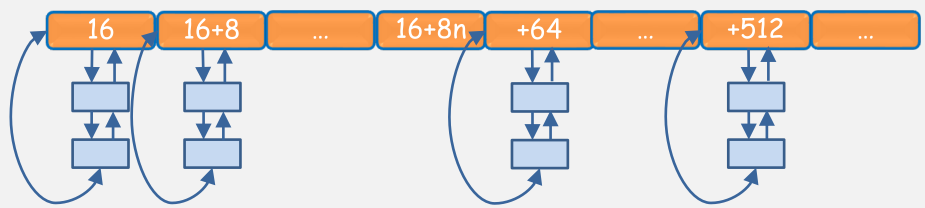
堆分配器：Doug Lea's Allocator (dlmalloc)

- 通过bins管理空闲内存块（chunks）
- 每个Regular bin是一个双向链表，包含大小固定的块
  - fastbin采用单向链表
- malloc()进行内存分配时需找到合适bin

```
Bins for sizes < 512 bytes contain chunks of all the same size,
spaced 8 bytes apart. Larger bins are approximately logarithmically
spaced:

    64 bins of size       8
    32 bins of size      64
    16 bins of size     512
     8 bins of size    4096
     4 bins of size   32768
     2 bins of size  262144
     1 bin  of size what's left
    The bins top out around 1MB because we expect to service large
    requests via mmap.
```

small bins

large bins



https://github.com/ennorehling/dlmalloc/blob/master/malloc.c

二、智能指针

思考如何在TeaPL中实现智能指针？

如何自动释放内存

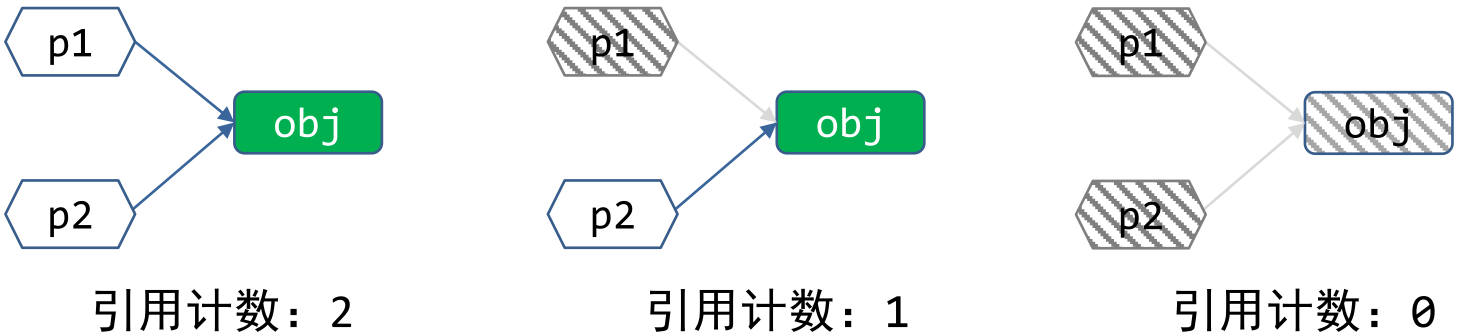
- 传统C/C++需要手动释放内存
  - malloc/free
  - constructor/destructor
- 如何自动释放内存？
  - 静态分析目标对象的生命周期：指针分析问题
  - 动态分析目标对象的引用数

```
//C++代码
int main() {
    Point x(1,1);
    Point* y = new Point(2,2);
    //delete y;//需要手动释放
}
```

```
Point (1,1) is dropped
```

动态分析记录引用数：智能指针

- 每产生一个新的引用，计数器加1，反之则减1
- 引用计数清零时自动释放资源



C++(11) 智能指针

- 独占型指针：unique_ptr
  - 通过move转移所有权
- 共享型指针：shared_ptr
  - 引用数为0时自动析构目标对象
  - 可以通过reset()主动释放引用数

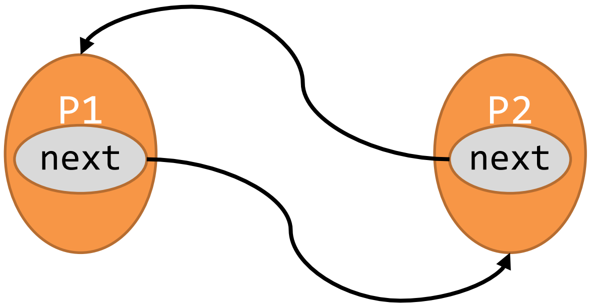
```
//C++代码
int main() {
    unique_ptr<MyClass> up1(new MyClass(2));
    //unique_ptr<MyClass> up2 = up1; //编译报错
    unique_ptr<MyClass> up2 = move(up1);
    //cout << up1->val << endl; //segmentation fault
    cout << up2->val << endl;

    shared_ptr<MyClass> sp1(new MyClass(2));
    shared_ptr<MyClass> sp2 = p1;
}
```

下面代码会输出什么？

```
//C++代码
class MyClass{
  public:
    int val;
    MyClass(int v) { val = v; }
    ~MyClass() { cout << "delete obj:"<< val << endl; }
};

int main() {
    MyClass* p0 = new MyClass(1);
    {
        shared_ptr<MyClass> p1(new MyClass(2));
        shared_ptr<MyClass> p2 = p1;
        shared_ptr<MyClass> p3(p0);
    }
    cout << p0->val << endl;
}
```

```
./a.out
delete obj:1
delete obj:2
0
```

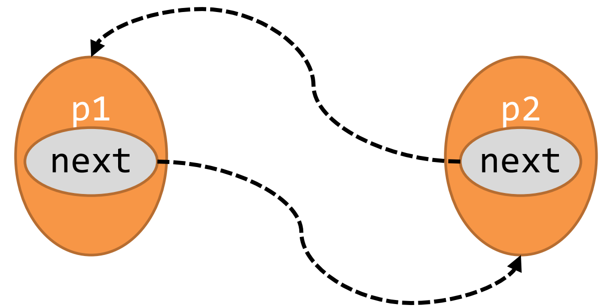
智能指针的主要问题：循环引用




```
//C++代码
class MyList{
public:
    int val;
    shared_ptr<MyList> next;
    ~MyList() { cout << "delete obj:"<< val << endl; }
};

int main() {
    shared_ptr<MyList> p1 = make_shared<MyList>();
    shared_ptr<MyList> p2 = make_shared<MyList>();
    p1->val = 1;
    p2->val = 2;
    p1->next = p2;
    p2->next = p1;
}
```

解决循环引用：weak_ptr

- 不改变引用计数




```
//C++代码
class MyList{
public:
    int val;
    weak_ptr<MyList> next;
    ~MyList() { cout << "delete obj:"<< val << endl; }
};

int main() {
    shared_ptr<MyList> p1 = make_shared<MyList>();
    shared_ptr<MyList> p2 = make_shared<MyList>();
    p1->val = 1;
    p2->val = 2;
    p1->next = p2;
    p2->next = p1;
}
```

三、代码复用和继承

结构体定义和代码复用

```
struct A {
    int a;
    float b;
}

struct B : A {
    int foo();
}
```

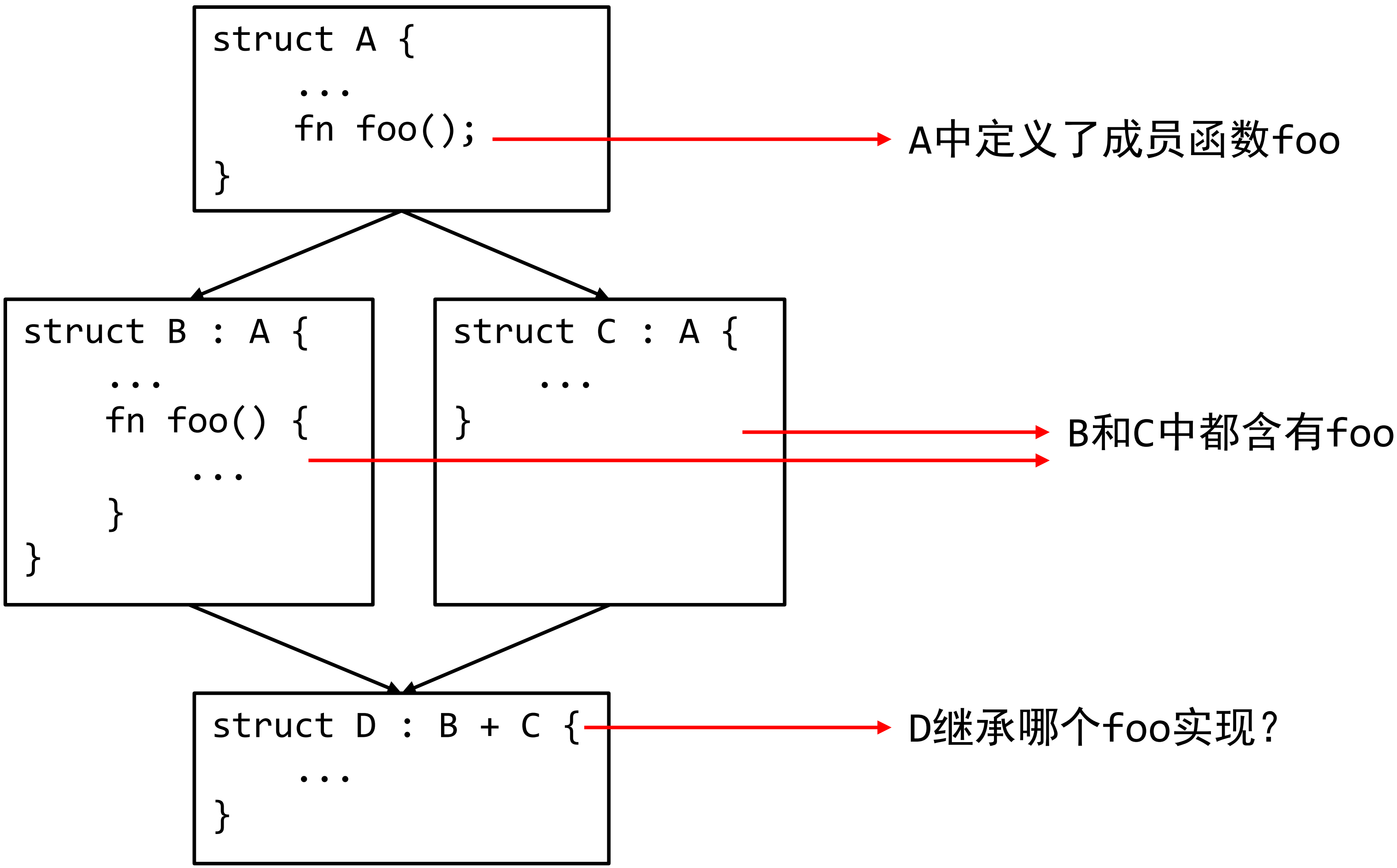
继承：结构体类型复用

```
struct A { … }
struct B { … }
struct S : B + A { …
}
```

继承多个结构体类型

如果A和B都包含同一个变量名或函数？

菱形继承问题


struct A {
...
fn foo();
}
A中定义了成员函数foo
struct B : A {
...
fn foo() {
...
}
}
struct C : A {
...
}
B和C中都含有foo
struct D : B + C {
...
}
D继承哪个foo实现？

应对多继承问题

```
class S {...}
interface A { fn foo(); }
interface B { fn bar(); }

impl A, B for S {
    fn foo(){...}
    fn bar(){...}
}
```

Java:不支持多继承

- 规格继承：Interface
- Interface只包括虚函数

```
class A {
    fn foo();
}
```

```
class B : virtual A {
    fn foo() { ... }
}
```

```
class C : virtual A {
    ...
}
```

```
class D : B, C {
    ...
}
```

C++：虚拟继承

功能代码复用：Mixin

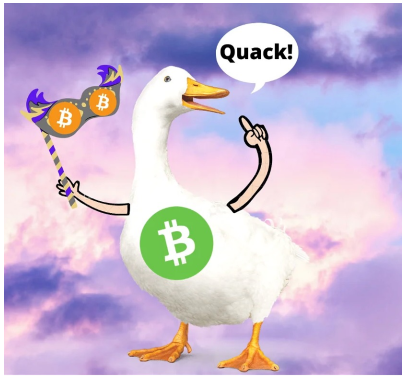
- Mixin：使用其它class中的方法而无需继承

```
//Java代码
interface A {
    fn foo();
}

interface B {
    fn bar();
}

class ImplA : impl A{
    fn foo(){...}
}

class ImplB : impl B{
    fn bar(){...}
}
```

```
//Java代码
class S impl A, B {
    ImplA a;
    ImplB b;

    fn foo(){
        a.foo();
    }

    fn bar(){
        b.bar();
    }
}
```

功能代码复用：Rust Trait

- Duck Typing：数据和功能分离的思想



“If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck”

```
//Rust代码
struct S {...}

trait A {
    fn foo(){...};
}

trait B : A {
    fn bar(){...};
}

impl B for S { }

struct S s;
s.foo();
s.bar();
```

声明struct S

定义trait A

定义trait B，继承A

为类型S实现trait B

S类型的变量可以调用A和B中的函数

四、子类型和协变

比较两个数的大小，并返回较大的一个

- 泛型参数：
  - C++ 模版（template）
  - Rust 泛型（generic）

```
//C++代码
int max(int x, int y) {
    return (x > y) ? x : y;
}
double max(double x, double y) {
    return (x > y) ? x : y;
}
char max(char x, char y) {
    return (x > y) ? x : y;
}
max(3, 7);
max(3.0, 7.0);
max('g', 'e');
```

```
//C++代码
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

```
//Rust 代码
fn max(x:T, y:T) -> T {
    return if(x > y) x else y;
}
```

多个泛型参数：C++

- 编译阶段推导确定具体类型
- 也可以通过属性指定泛型的具体类型

```
//C++代码
template <typename T, typename G>
auto max(T x, G y) {
    return (x > y) ? x : y;
}

max(3, 7);                →define i32 @maxi32i32(i32 %0, i32 %1)
max(3.0, 7.0);            →define i32 @maxf32f32(f32 %0, f32 %1)
max<int,char>(3, 'g');    →define i32 @maxi32i8(i32 %0, i8 %1)
max(3, 7.0);              →define i32 @maxi32f32(i32 %0, f32 %1)
max(3.0, 'g');            →define i32 @maxf32i8 (f32 %0, i8 %1)
max('g', 3);              →define i32 @maxi8i32(i8 %0, i32 %1)
```

https://en.cppreference.com/w/cpp/language/template_argument_deduction

子类型

- 类型之间存在偏序关系，如X≤Y表示：
  - X是Y的子类型
  - Y是的父类型
- 偏序的特性：
  - 自反性：X≤X
  - 传递性：X≤Y，Y≤Z $\Longrightarrow$ X≤Z

Liskov替换原理和类型约束

- 当类型约束为父类型时，可用子类型的对象
- 子类型的数据结构可兼容父类型

```
//Java代码
public class B extends Number {
    ...
}

public class A {
    public <T extends Number> void foo(T t){
        ...
    }
}

class A a;
class B b;
a.foo(b);
```

Upcast和Downcast

- Upcasting：如果X>Y，将Y类型转换为X类型
  - Liskov替换原理：一般不存在风险，默认都允许
- Downcasting：如果X>Y，将X类型转换为Y类型
  - 类型检查，如果类型不匹配会抛出异常

```
//C++代码
class Base {};
class Derived : public Base {};

int main(int argc, const char** argv) {
    Base* base = new Base();
    if(Derived* derived = dynamic_cast<Derived *>(base)){
        ...
    }
}
```

Trait之间可以存在偏序关系

- 但非类型之间的偏序关系

```
//Rust代码
struct S { }
trait A { }
trait B : A { }
impl B for S { }
```

=> B < A

```
//Rust代码
trait A { }
trait B { }
impl<T> B for T where T:A { }
```

=> B < A

```
//Rust代码
struct S1 { }
struct S2 { }
trait A { }
trait B { }
impl A for S2 { }
impl B for S2 { }
impl A for S1 { }
```

=> S2 < S1

Trait用于类型约束

```
//Rust代码
trait A { }
trait B : A { }
struct S { }

impl A for S { }
impl B for S { }

fn makeacall<T:A>(s: &T){
    ...
}

fn main() {
  let a = S {};
  makeacall(&a);
}
```

```
//Rust代码
trait A { }
trait B { }
struct S { }

impl A for S { }
impl<T> B for T where T:A { }

fn makeacall<T:B>(s: &T){
    ...
}

fn main() {
  let a = S {};
  makeacall(&a);
}
```

Liskov替换带来的问题

```
class A {
    fn foo();
}
```

```
class B : A {
    fn foo() {
        ...
    }
}
```

```
class C : A {
    ...
}
```

B和C都是A的子类

```
fn makeacall(a : A) {
    a.foo();
}
B b;
C c;
makeacall(b);
makeacall(c);
```

参数要求：A或A的子类

调用哪个foo函数实现？

下面这段C++代码输出什么？

```
class Base { //C++代码
public:
  void print(){ cout << "base print" << endl;}
  virtual void speak(){ cout << "base speak" << endl;}
  virtual void shout(){ cout << "base shout" << endl;}
  virtual ~Base(){ cout << "destroying base" << endl;}
};

class Derived : public Base {
public:
  void print(){ cout << "derived print" << endl;}
  virtual void speak(){ cout << "derived speak" << endl;}
  virtual ~Derived(){ cout << "destroying derived" << endl;}
};

void test(Base* bptr){
  bptr->print();
  bptr->speak();
  bptr->shout();
}

int main(){
  Derived dobj;
  test(&dobj);
}
```

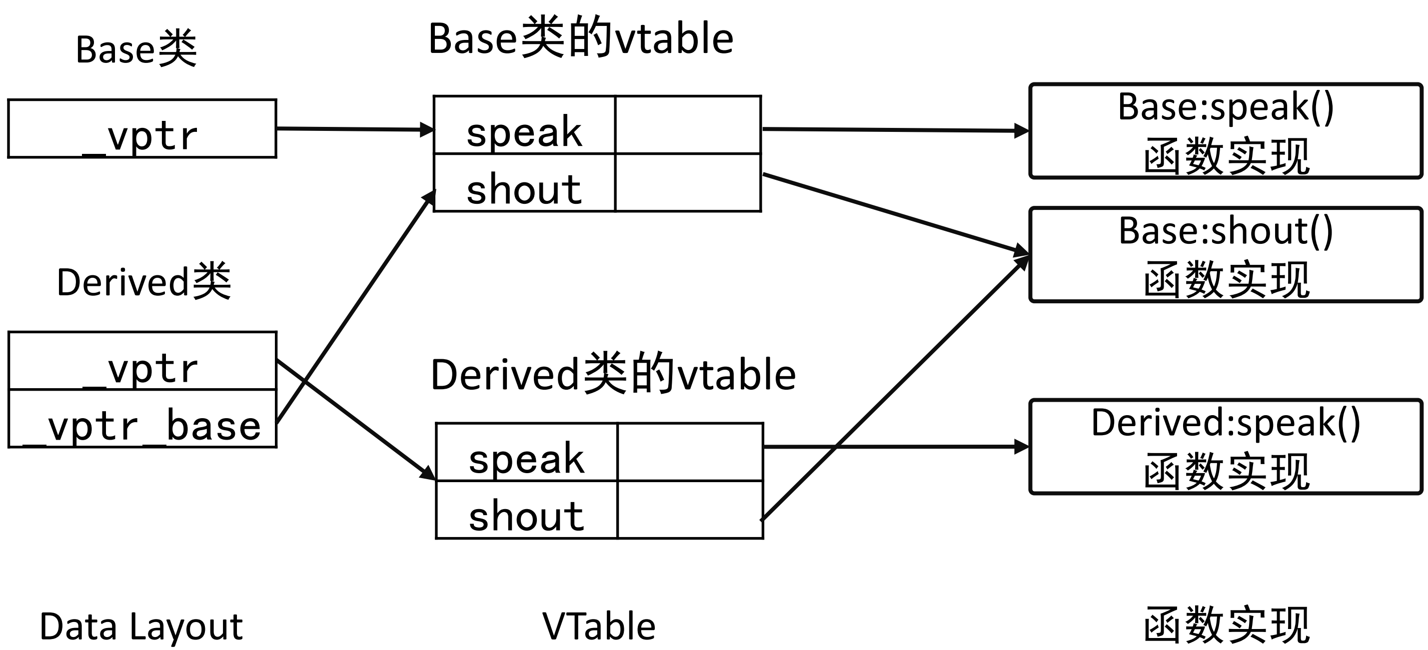
```
base print
derived speak
base shout
destroying derived
```

虚函数和动态绑定

- 静态绑定：在编译时确定执行版本
  - 通过对象类型调用任意函数
  - 调用实函数
- 动态绑定：直到运行时才能确定执行版本
  - C++虚函数
  - Rust dynamic trait

C++如何实现动态分发

- 编译器为每个类创建一个虚拟指针（vptr）指向虚拟方法表格（vtable：virtual method table）
- vtable包含每一个可用虚函数以及指向其具体函数实现的指针。



https://itanium-cxx-abi.github.io/cxx-abi/cxx-vtable-ex.html

IR表示

```
%class.Derived = type { %class.Base }
%class.Base = type { ptr }
```

```
//C++代码
void test(Base* bptr){
  bptr->print();
  bptr->speak();
  bptr->shout();
}
```

```
define void @test(ptr %0) {
  %2 = alloca ptr
  store ptr %0, ptr %2
  %3 = load ptr, ptr %2
  call void @_ZN4Base5printEv(ptr %3)
  %4 = load ptr, ptr %2
  %5 = load ptr, ptr %4
  %6 = getelementptr ptr, ptr %5, i64 0
  %7 = load ptr, ptr %6
  call void %7(ptr %4)
  %8 = load ptr, ptr %2
  %9 = load ptr, ptr %8
  %10 = getelementptr ptr, ptr %9, i64 1
  %11 = load ptr, ptr %10
  call void %11(ptr %8)
  ret void
}
```

```
Derived dobj;
test(&dobj);
```

```
%1 = alloca %class.Derived
%3 = alloca i32
call void @_ZN7DerivedC2Ev(ptr %1)
invoke void @_Z4testP4Base(ptr %1)
```

子类型关系是否会自动传播？

- 如果A是B的子类型，那么
  - A型数组和B型数组的关系？
  - List<A>和List<B>呢？

```
//Rust代码
fn foo(b:&[B]){ ... }
fn foo(l:List<B>){ ... }
```

协变关系：covariance

- 如果A是B的子类型，T<A>是T<B>的子类型
- 可能会引入错误，需要动态类型检查

```
//Java代码
String[] a = new String[1];
Object[] b = a;
b[0] = 1;
```

运行时报错

逆变关系：contravariance

- 如果A是B的子类型，T<A>是T<B>的子类型
- 典型逆变关系：函数参数

```
//Rust代码
fn test(f:fn(A)->()){
    ...
}
fn foo(a:A) { ... }
fn bar(b:B) { ... }
test(bar)
```

bar是foo的子类型

五、函数式编程

函数式编程的特性

- 函数是一等公民：可用作变量赋值、参数传递、返回值
- 高阶函数：如y=f(g(x))
- 在命令式编程语言中：
  - C++ lambda表达式
  - Rust closure

C++ Lambda表达式

- 延迟计算
- 环境变量：[]；参数传递：()

```
//C++代码
template <typename F>
int hofn(int v1, int v2, F f) {
    return f(v1, v2);
}

int main() {
    int i = 10;
    auto cl = [i](int a, int b) {
                std::cout << "In closure" << std::endl;
                return a + b + i;
            };
    std::cout << "After closure" << std::endl;
    int result = hofn(20, 10, cl);
    std::cout << "Result: " << result << std::endl;

    return 0;
}
```

a和b是参数，i是捕获变量

```
./a.out
After closure
In closure
Result: 40
```

Rust Closure

- 自动捕获环境变量
- 参数传递：||

```
//Rust代码
fn hofn<F>(v1:i32, v2:i32, f: F) -> i32
    where F: Fn(i32, i32) -> i32
{
    f(v1,v2)
}

fn main() {
    let i = 10;
    let cl = move |a, b| { a+b+i };
    let result = hofn(20, 10, cl);
}
```

a和b是参数，i是捕获变量

使用函数作返回值

```
//Rust代码
fn hofn(len:u32) -> Box<dyn Fn(u32) -> u32> {
    let vec:Vec<u32> = (1..len).collect();
    let sum:u32 = vec.iter().sum();
    Box::new(move |x| {
        sum + x
    })
}

fn main() {
    hofn(10)(10);
}
```

Monad

- 将返回值封装在含有功能代码的结构体中

```
//Rust代码
enum Result<T, E> {
   Ok(T),
   Err(E),
}

fn foo(v:i32)
   -> Result<i32, &'static str> {
    match v {
        0 => Err("invalid"),
        _ => Ok(v),
    }
}

let r = foo(1);
match r {
    Ok(v) => ...,
    Err(e) => println!("{e:?}"),
}
```

```
//Rust代码
pub enum Option<T> {
    None,
    Some(T),
}

fn foo(v: int) -> Option<int> {
    match v {
        0 => None,
        _ => Some(v)
    }
}

let x = foo(1);
let y = match x {
    Some(x) => x,
    None    => 0,
};
```

高阶函数典型应用

- 通过Iterator实现容器的filter、map等功能

```
//Rust代码
fn main() {
    let mut v:Vec<u32> = (1..100).collect();
    let iter1 = v.iter();
    let sum1:u32 = iter1().sum();

    let iter2 = v.iter().filter(|x| *x % 2 as u32 == 0);
    let sum2:u32 = iter2().sum();
    println!("sum = {:?}{:?}", sum1, sum2);

    let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();
    println!(”v2 = {:?}", v2);
}
```

Iterator的优点

- 循环会做两次边界检测
  - 循环条件检查
  - 越界检查
- Iterator只检查一次

```
//Rust代码
let len = 1000000;
let mut vec:Vec<usize> = (1..len).collect();
let start = Instant::now();
for i in vec.iter_mut(){
    *i += 1;
}
println!("{:?}", start.elapsed().as_nanos());

let start = Instant::now();
for i in 0..len-1 {
    vec[i] = vec[i]-1;
}
println!("{:?}", start.elapsed().as_nanos());
```

```
#:./a.out
14253222
57399993
```