

Pointers

指標就是力量

scc@teamt5.org

Outline

- The C World
 - 頭腦體操
 - Array
 - Compile time check
 - container_of
 - restrict qualifier
- The C++ World
 - Ownership
 - Virtuals and this pointer

The C world

頭腦體操

```
int **ptr;  
int *(*ptr)[10];  
int *(*ptr)(int *);  
int *(*ptr[10])(int **);  
int (*( *(*ptr(int *))[10]))(int *);  
sizeof(int (*****)[10][20][30][40]);
```

頭腦體操

```
int **ptr;
```

```
int *(*ptr)[10];
```

```
int *(*ptr)(int *);
```

```
int *(*ptr[10])(int **);
```

```
int (*( *(*ptr(int *))[10]))(int *);
```

```
sizeof(int (*****)[10][20][30][40]);
```

pointer to pointer

頭腦體操

```
int **ptr;
```

```
int *(*ptr)[10];
```

```
int *(*ptr)(int *);
```

```
int *(*ptr[10])(int **);
```

```
int (*( *(*ptr(int *))[10]))(int *);
```

```
sizeof(int (*****)[10][20][30][40]);
```

pointer to array 10 of pointer to char

頭腦體操

```
int **ptr;  
int *(*ptr)[10];  
int *(*ptr)(int *);  
int *(*ptr[10])(int **);  
int (*( *(*ptr(int *))[10]))(int *);  
sizeof(int (*****)[10][20][30][40]);
```

function (pointer to int) returning
pointer to int

頭腦體操

```
int **ptr;  
int *(*ptr)[10];  
int *(*ptr)(int *);  
int *(*ptr[10])(int **);  
int (*( *(*ptr(int *))[10]))(int *);  
sizeof(int (*****)[10][20][30][40]);
```

array 10 of pointers to function
(pointer to pointer to int) returning a
pointer to int

頭腦體操

```
int **ptr;  
int *(*ptr)[10];  
int *(*ptr)(int *);  
int *(*ptr[10])(int **);  
int (*( *(*ptr(int *))[10]))(int *);
```

```
sizeof(int (*****)[10][20][30][40]);
```

pointer to pointer to function (pointer
to int) returning array 10 of function
(pointer to int) returning pointer to int

cdecl

C gibberish ↔ English

declare ptr as pointer to pointer to function (pointer to int) returning array 10 of

```
int *(*(*ptr)(int *))[10](int *)
```

[permalink](#)

頭腦體操

```
int **ptr;  
int *(*ptr)[10];  
int *(*ptr)(int *);  
int *(*ptr[10])(int **);  
int (*( *(*ptr(int *))[10]))(int *);  
sizeof(int (*****)[10][20][30][40]);
```

pointer to pointer to pointer to
pointer to pointer to pointer to
array of 10 array of 20 array of 30
array of 40 int

頭腦體操

```
int **ptr;  
int *(*ptr)[10];  
int *(*ptr)(int *);  
int *(*ptr[10])(int **);  
int (*( *(*ptr(int *))[10]))(int *);  
sizeof(int (*****)[10][20][30][40]);
```

pointer to pointer to pointer to
pointer to pointer to pointer to
array of 10 array of 20 array of 30
array of 40 int

Anyway, it's also a pointer.

Array



Jyun-An Chen

11月16日上午4:54 · 🌐

現在是凌晨 4 點 54 分

我來分享一個 C/C++ 冷知識

陣列 `name[index]` 的寫法可以寫成 `index[name]`

如圖片，我整個人都嚇傻了

```
5 int main()
6 {
7     int a[5] = { 10 , 20 , 30 , 40 , 50 };
8
9     cout << 3[a] << "\n"; // Output: 40
10 }
```



馬聖豪、蔡孟師和其他108人

41則留言 37次分享

6.5.2.1 Array subscripting

Constraints

- 1 One of the expressions shall have type “pointer to object *type*”, the other expression shall have integer type, and the result has type “*type*”.

Semantics

- 2 A postfix expression followed by an expression in square brackets `[]` is a subscripted designation of an element of an array object. The definition of the subscript operator `[]` is that `E1[E2]` is identical to `(*(E1 + E2))`. Because of the conversion rules that apply to the binary `+` operator, if `E1` is an array object (equivalently, a pointer to the initial element of an array object) and `E2` is an integer, `E1[E2]` designates the `E2`-th element of `E1` (counting from zero).

6.5.2.1 Array subscripting

Constraints

1 One of the expressions shall have type “pointer to object type” the other expression shall have integer type, and the result has type “type

$\text{sizeof}(\mathbf{E1}) * \mathbf{E2}$

Semantics

2 A postfix expression followed by an expression in square brackets is a subscripted designation of an element of an array object. The definition of the subscript operator `[]` is that $\mathbf{E1}[\mathbf{E2}]$ is identical to $(*(\mathbf{E1} + \mathbf{E2}))$. Because of the conversion rules that apply to the binary `+` operator, if $\mathbf{E1}$ is an array object (equivalently, a pointer to the initial element of an array object) and $\mathbf{E2}$ is an integer, $\mathbf{E1}[\mathbf{E2}]$ designates the $\mathbf{E2}$ -th element of $\mathbf{E1}$ (counting from zero).

Array

6.5.2.1 Array

Constraints

1 One of the expressions shall have integer type.

Semantics

2 A postfix expression followed by an expression in square brackets designates an element of an array object. The definition of the subscripting operator `[]` is identical to `(*((E1) + (E2)))`. Because of the conversion rules that apply to the `+` operator, if **E1** is an array object (equivalently, a pointer to the first element of an array object) and **E2** is an integer, **E1[E2]** designates the **E2**-th element of **E1** (counting from zero).

挑戰題: **++ptr[--ptr]**

Which one is sequenced before?

`sizeof(E1) - E2`

Array

挑戰題: **`++ptr[--ptr]`**

7.6.1.1 Subscripting

[**`expr.sub`**]

- ¹ A postfix expression followed by an expression in square brackets is a postfix expression. One of the expressions shall be a glvalue of type “array of T” or a prvalue of type “pointer to T” and the other shall be a prvalue of unscoped enumeration or integral type. The result is of type “T”. The type “T” shall be a completely-defined object type.⁵⁸ The expression `E1[E2]` is identical (by definition) to `*((E1)+(E2))`, except that in the case of an array operand, the result is an lvalue if that operand is an lvalue and an xvalue otherwise. The expression `E1` is sequenced before the expression `E2`.



UB



Well-defined
after 17

Indirect pointers

void

}



<https://www.youtube.com/watch?v=o8NPllzkFhE>

Indirect pointers

```
void remove_list_node(List *list, Node *target) {
    Node *prev = NULL;
    Node *current = list->head;
    // Walk the list
    while (current != target) {
        prev = current;
        current = current->next;
    }
    // Remove the target by updating the head or the previous node.
    if (!prev)
        list->head = target->next;
    else
        prev->next = target->next;
}
```

Indirect pointers

```
void remove_list_node(List *list, Node *target) {  
    // The "indirect" pointer points to the *address*  
    // of the thing we'll update.  
    Node **indirect = &list->head;  
    // Walk the list, looking for the thing that  
    // points to the node we want to remove.  
    while (*indirect != target)  
        indirect = &(*indirect)->next;  
    *indirect = target->next;  
}
```

callback and member function

```
int (*callback)(char *) = info_getter[ctz];  
if (callback == NULL) {  
    // It might impossible to reach here, but just in case  
    pr_alert("Unsupported info: %d\n", ctz);  
    return -EINTR;  
}  
  
int offset = callback(info);  
  
// A dispatcher table for those callbacks  
const static int (*info_getter[ALIGN(NUM)])(char *) = {  
    f1,  
    f2,  
    f3,  
    f4,  
    f5,  
    f6,  
    NULL,  
};
```

callback and member function

```
const struct file_operations fops = {  
    .owner = THIS_MODULE,  
    .read = my_read,  
    .write = my_write,  
    .open = my_open,  
    .release = my_release,  
    .llseek = my_device_llseek,  
};
```

```
struct file_operations {  
    struct module *owner;  
    loff_t (*llseek) (struct file *, loff_t, int);  
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);  
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);  
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);  
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);  
    //...  
} __randomize_layout;
```

NULL for compiler check?

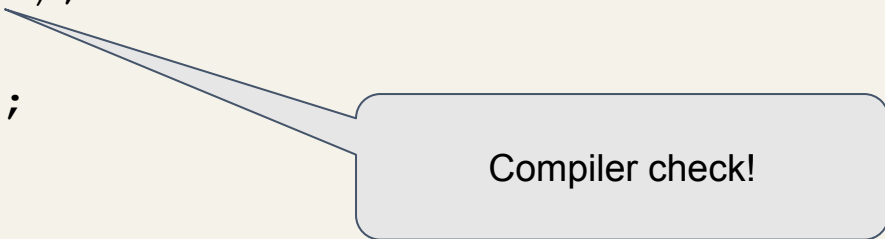
```
int foo(const int arr[static 1]) {}
```

```
int main() {  
    foo(NULL);  
    int a;  
    foo(&a);  
}
```

NULL for compiler check?

```
int foo(const int arr[static 1]) {}
```

```
int main() {  
    foo(NULL);  
    int a;  
    foo(&a);  
}
```



Compiler check!

NULL for compiler check?

C99: §6.7.5.3/7

```
int foo(const int  
  
int main() {  
    foo(NULL);  
    int a;  
    foo(&a);  
}
```

A declaration of a parameter as “array of *type*” shall be adjusted to “qualified pointer to *type*”, where the type qualifiers (if any) are those specified within the [and] of the array type derivation. If the keyword **static** also appears within the [and] of the array type derivation, then for each call to the function, the value of the corresponding actual argument shall provide access to the first element of an array with at least as many elements as specified by the size expression.

```
<source>: In function 'main':
```

```
<source>:8:5: warning: argument 1 null where non-null expected [-Wnonnull]
```

```
8 |     foo(NULL);  
  |     ~~~
```

```
<source>:3:5: note: in a call to function 'foo' declared 'nonnull'
```

```
3 | int foo(const int arr[static 1]) {  
  |     ~~~
```


The void pointer

This is the only one way to implement containers in CS101.

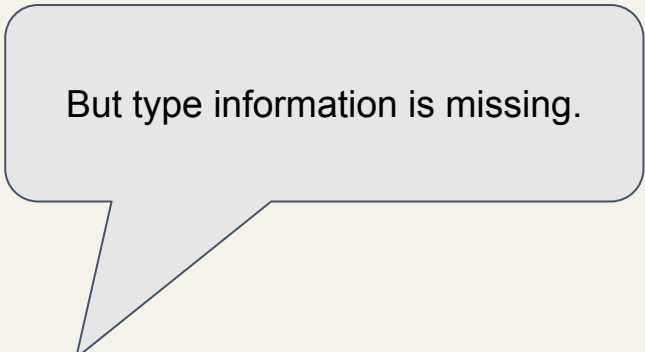
The void pointer

```
typedef struct vector {  
    void *data;  
    unsigned int size;  
} vector_t;  
  
void* push_back(vector_t* self, void *item) {  
    //...  
}
```

The void pointer

```
typedef struct vector {  
    void *data;  
    unsigned int size;  
} vector_t;
```

```
void* push_back(vector_t* self, void *item) {  
    //...  
}
```



But type information is missing.

The void pointer

```
#define VECTOR_PUSH_BACK(v, item, type) \
do { \
    if ((v).size >= (v).capacity) { \
        (v).capacity *= 2; \
        void *temp = realloc((v).data, (v).capacity * (v).item_size); \
        if (temp == NULL) { \
            fprintf(stderr, "Memory reallocation failed\n"); \
            exit(EXIT_FAILURE); \
        } \
        (v).data = temp; \
    } \
    memcpy((char *) (v).data + ((v).size * (v).item_size), &item, \
(v).item_size); \
    (v).size++; \
} while (0)
```

The void pointer

```
#define VEC  
do {  
    if (  
  
    }  
    memc  
(v).item_si  
(v).  
} while
```

```
45 int main() {  
46     vector_t myVector;  
47     VECTOR_INIT(myVector, int); // Initialize vector for integers  
48  
49     int value1 = 10;  
50     int value2 = 20;  
51  
52     VECTOR_PUSH_BACK(myVector, value1, int); // Push value1  
53     VECTOR_PUSH_BACK(myVector, value2, int); // Push value2  
54  
55     // Access elements  
56     int *arr = (int *)myVector.data;  
57     for (size_t i = 0; i < myVector.size; ++i) {  
58         printf("%d ", arr[i]);  
59     }  
60     printf("\n");  
61  
62     free(myVector.data); // Free allocated memory  
63     return 0;  
64 }
```

The void pointer

```
45 int main() {
46     vector_t myVector;
47     VECTOR_INIT(myVector, int); // Initialize vector for integers
48
49     int value1 = 10;
50     int value2 = 20;
51
52     VECTOR_PUSH_BACK(myVector, value1, int); // Push value1
53     VECTOR_PUSH_BACK(myVector, value2, int); // Push value2
54
55     // Access elements
56     int *arr = (int *)myVector.data;
57     for (size_t i = 0; i < myVector.size; ++i) {
58         printf("%d ", arr[i]);
59     }
60     printf("\n");
61
62     free(myVector.data); // Free allocated memory
63     return 0;
64 }
```

```
#define VEC
do {
    if (
memc
(v).item_si
(v).
} while
```

The void pointer

How's about
container in a
container?

```
45 int main() {
46     vector_t myVector;
47     VECTOR_INIT(myVector, int); // Initialize vector for integers
48
49     value1 = 10;
50     value2 = 20;
51
52     PUSH_BACK(myVector, value1, int); // Push value1
53     PUSH_BACK(myVector, value2, int); // Push value2
54
55     // Access elements
56     int *arr = (int *)myVector.data;
57     for (size_t i = 0; i < myVector.size; ++i) {
58         printf("%d ", arr[i]);
59     }
60     printf("\n");
61
62     free(myVector.data); // Free allocated memory
63     return 0;
64 }
```

The void pointer

How's about
container in a
container?

It's hard to implement:
`vector_t(vector_t(double), int)...`

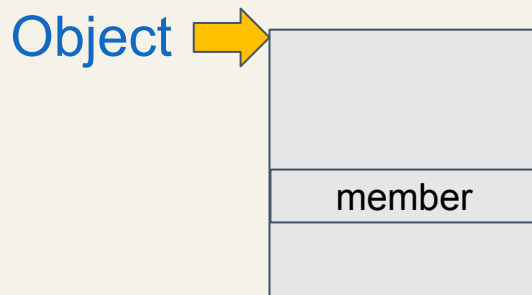
```
45 int main() {  
46     vector_t myVector;  
47     VECTOR_INIT(myVector, int); // Initialize vector for integers  
48  
49     int value1 = 10;  
50     int value2 = 20;  
51  
52     PUSH_BACK(myVector, value1, int); // Push value1  
53     PUSH_BACK(myVector, value2, int); // Push value2  
54  
55     // Access  
56     int *arr  
57     for (si  
58     pri  
59  
60     printf(  
61  
62     free(myVector.data); // Free allocated memory  
63     return 0;  
64 }  
65
```


container_of

```
#define container_of(ptr, type, member) \
__extension__({ \
    const __typeof__(((type *)0)->member) *(__pmember) = (ptr); \
    (type *)((char *)__pmember - offsetof(type, member)); \
})
```

container_of

```
#define container_of(ptr, Object, member) \
__extension__({ \
    const __typeof__(((Object *)0)->member) *(__pmember) = (ptr); \
    (type *)((char *)__pmember - offsetof(type, member)); \
})
```



container_of

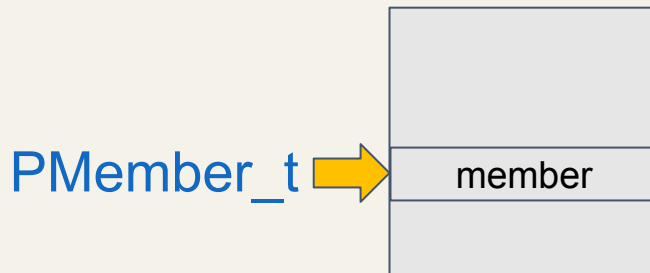
```
#define container_of(ptr, type, member) \
__extension__({ \
    const __typeof__(((type *)0)->member) *(__pmember) = (ptr); \
    (type *)((char *)__pmember - offsetof(type, member)); \
})
```

PMember



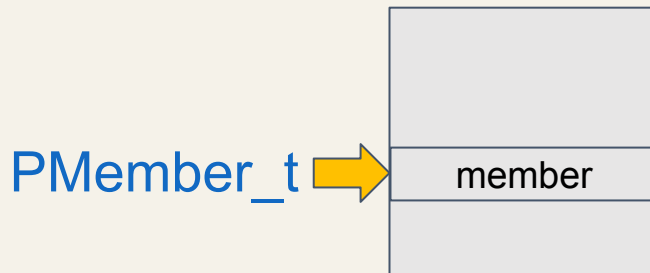
container_of

```
#define container_of(ptr, type, member) \
__extension__({ \
    const __typeof__(((type *)0)->member) *(__pmember) = (ptr); \
    (type *)((char *)__pmember - offsetof(type, member)); \
})
```



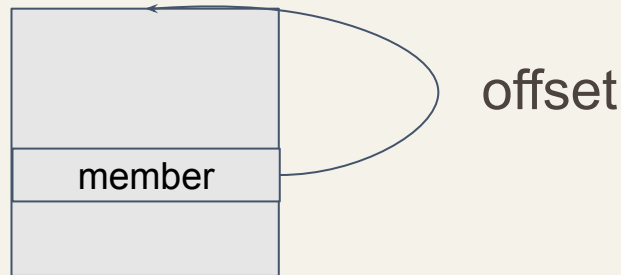
container_of

```
#define container_of(ptr, type, member) \
__extension__({ \
    const __typeof__(((type *)0)->member) *(__pmember) = (ptr); \
    (type *)((char *)__pmember - offsetof(type, member)); \
})
```



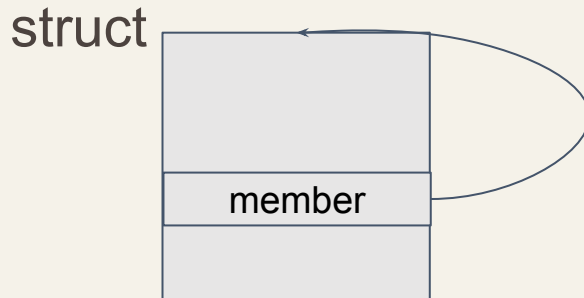
container_of

```
#define container_of(ptr, type, member) \
__extension__({ \
    const __typeof__(((type *)0)->member) *(__pmember) = (ptr); \
    (type *)((char *)__pmember - offsetof(type, member)); \
})
```



container_of


```
#define container_of(ptr, type, member) \
__extension__({ \
    const __typeof__(((type *)0)->member) *(__pmember) = (ptr); \
    (type *)((char *)__pmember - offsetof(type, member)); \
})
```



container_of

```
#define container_of(ptr, type, member) \
__extension__({ \
    const __typeof__(((type *)0)->member) *(__pmember) = (ptr); \
    (type *)((char *)__pmember - offsetof(type, member)); \
})
```

```
typedef struct {  
    char *value;  
    struct list_head list;  
} my_data;
```



container_of

```
#define container_of(ptr, type, member) \
__extension__({ \
    const __typeof__(((type *)0)->member) *(__pmember) = (ptr); \
    (type *)((char *)__pmember - offsetof(type, member)); \
})
```

Take care!

container_of

```
list_head_t *l = new_my_data(42);
```

```
const int me = get_value(l);
```

```
printf("%d\n", me);
```

```
list_head_t l2 = *l;
```

```
const int me2 = get_value(&l2);
```

```
printf("%d\n", me2);
```

Program returned: 0

42

-1947187831

container_of

```
list_head_t *l = new_my_data(42);
```

```
const int me = get_value(l);
```

```
printf("%d\n", me);
```

```
list_head_t l2 = *l;
```

```
const int me2 = get_value(&l2);
```

```
printf("%d\n", me2);
```

Local variable.

Program returned: 0

42

-1947187831

container_of

```
list_head_t *l = new_my_data(42);
```

```
const int me = get_value(l);
```

```
printf("%d\n", me);
```

```
list_head_t l2 = *l;
```

```
const int me2 = get_value(&l2);
```

```
printf("%d\n", me2);
```

Get offset from stack.

Program returned: 0

42

-1947187831

Extra:

Is container_of
well-defined?

restrict qualifier

- Not in C++
- For pointer optimizations
- since C99

restrict qualifier

```
void foo(int *a, int *b) {  
    *b = NULL;  
    *a = NULL;  
}
```

restrict qualifier

Any problem?

```
void foo(int *a, int *b) {  
    *b = NULL;  
    *a = NULL;  
}
```



restrict qualifier

What if `a == b`?

```
void foo(int *a, int *b) {
```

```
    *b = NULL;
```

```
    *a = NULL;
```

```
}
```



restrict qualifier

What if `a == b`?

```
void foo(int *a, int *b) {
```

```
    *b = NULL;
```

```
    *a = NULL;
```

```
}
```

Memory
Access
Violation

restrict qualifier

```
void foo(int *a, int *b) {  
    *b = NULL;  
    if (b != a)  
        *a = NULL;  
}
```

Extra branch!

restrict qualifier

User guarantee

```
void foo(int *a, int *b) {  
    *b = NULL;  
    if (b != a)  
        *a = NULL;  
}
```

Extra branch!

restrict qualifier

```
void foo(int *restrict a,  
         int *restrict b) {  
    *b = NULL;  
    *a = NULL;  
}
```

Compiler warning when
it is same pointer.

The C++ world

Ownership

RAII

Resource Acquisition Is Initialization

Ownership

```
int main() {  
    auto ptr = std::make_unique<int>(42);  
}
```

Ownership

```
int main() {  
    auto ptr = std::make_unique<int>(42);  
}
```



Release resources when destruct.

Ownership

This object occupies the ownership.

```
int main() {  
    auto ptr = std::make_unique<int>(42);  
}
```

Release resources when destruct.

Object lifetime


```
int main() {  
    auto ptr = std::make_unique<int>(42);  
}
```



Release resources when destruct.


Object lifetime

```
int main() {  
    auto ptr = std::make_shared<int>(42);  
}
```



Object lifetime

Reference count



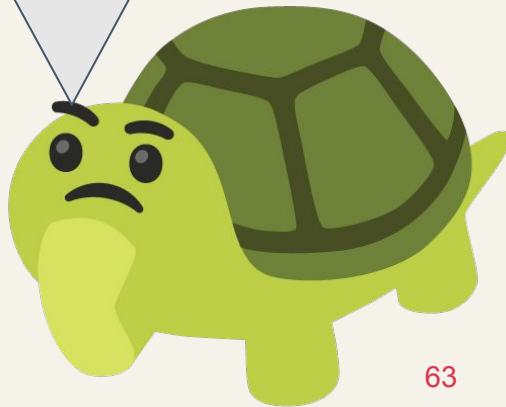
```
int main() {  
    auto ptr = std::make_shared<int>(42);  
}
```

Object lifetime

```
int main() {  
    auto ptr = std::make_shared<int>(42);  
}
```



std::atomic<std::shared_ptr>
Specialization since C++20.

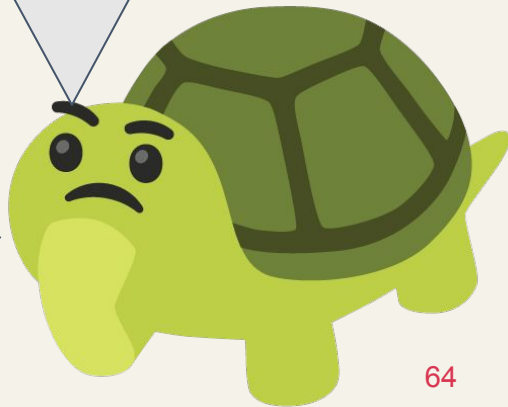


Object lifetime

```
int main() {  
    auto ptr = std::make_shared<int>(42);  
}
```

It's atomic operation
for **control block**.
But for user data.

std::atomic<std::shared_ptr>
Specialization since C++20.

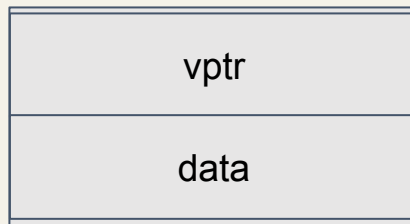


Virtual functions

```
struct Obj {  
    void normal() {}  
    virtual void foo() {}  
    virtual ~Obj = default;  
    int data;  
};
```

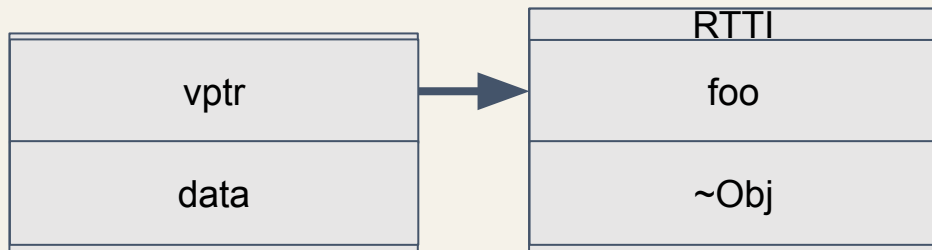
Virtual functions

```
struct Obj {  
    void normal() {}  
    virtual void foo() {}  
    virtual ~Obj = default;  
    int data;  
};
```



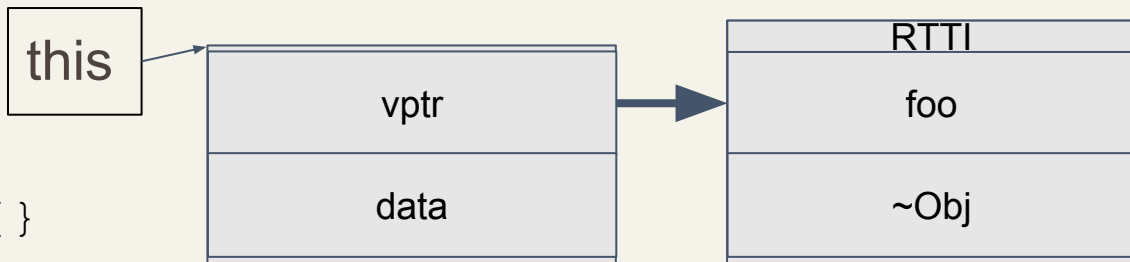
Virtual functions

```
struct Obj {  
    void normal() {}  
    virtual void foo() {}  
    virtual ~Obj = default;  
    int data;  
};
```



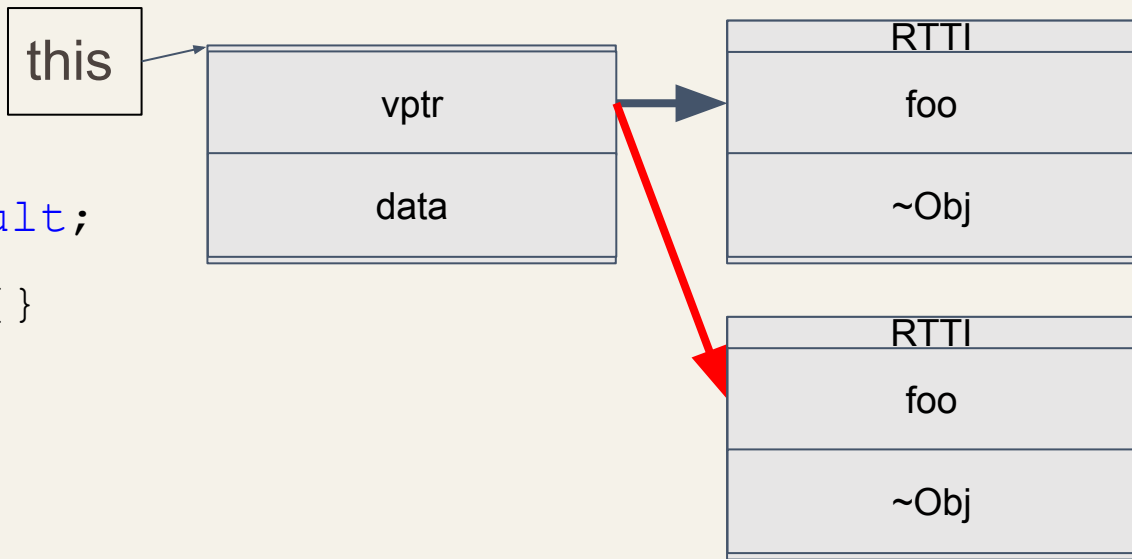
Virtual functions

```
struct Obj {  
    void normal() {}  
    virtual void foo() {}  
    virtual ~Obj = default;  
    int data;  
};
```

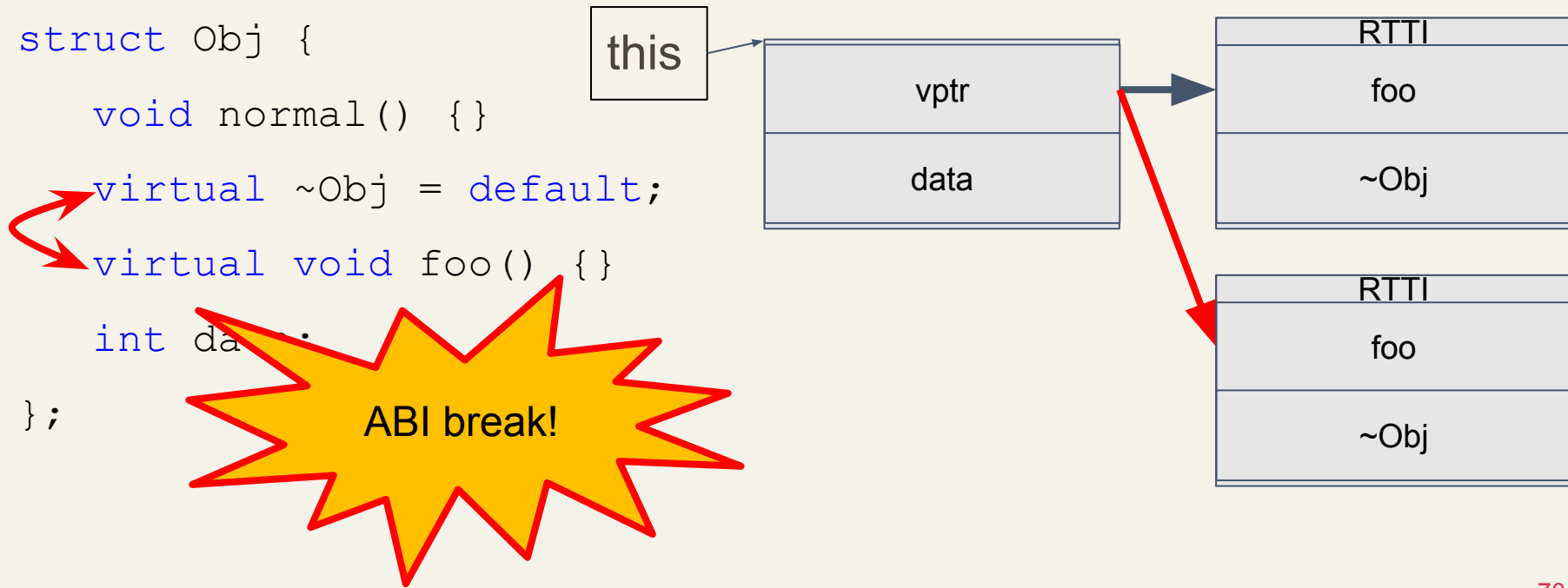


Virtual functions

```
struct Obj {  
    void normal() {}  
    virtual ~Obj = default;  
    virtual void foo() {}  
    int data;  
};
```

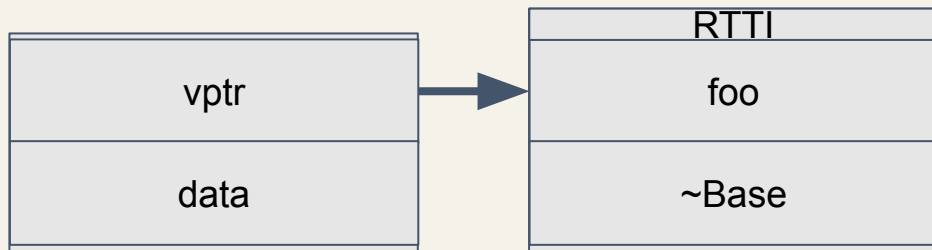


Virtual functions



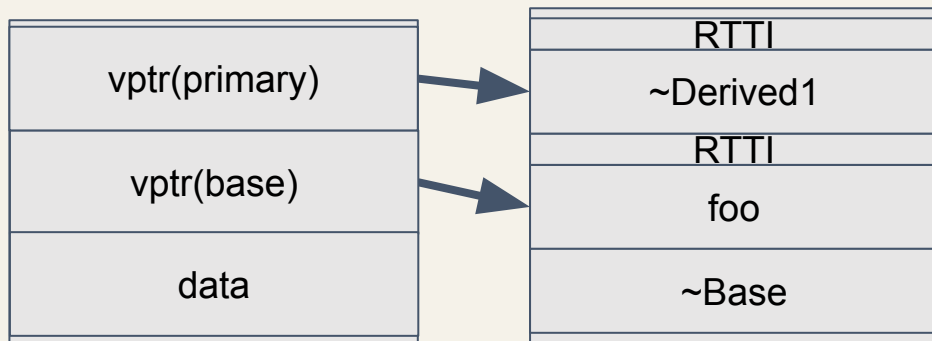
Virtual functions

```
struct Base {  
    virtual void foo() {}  
    virtual ~Base() = default;  
    int data;  
};
```



Virtual functions

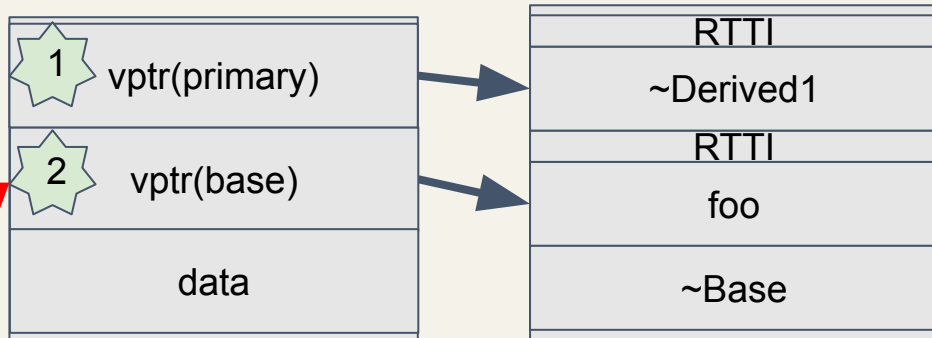
```
struct Base {  
    virtual void foo() {}  
    virtual ~Base() = default;  
    int data;  
};
```



```
struct Derived1 : virtual public Base { ~Derived1() = default; };  
struct Derived2 : virtual public Base { ~Derived2() = default; };
```


Virtual functions

```
struct Base {  
    virtual void foo() {}  
    virtual ~Base() = default;  
    int data;  
};
```

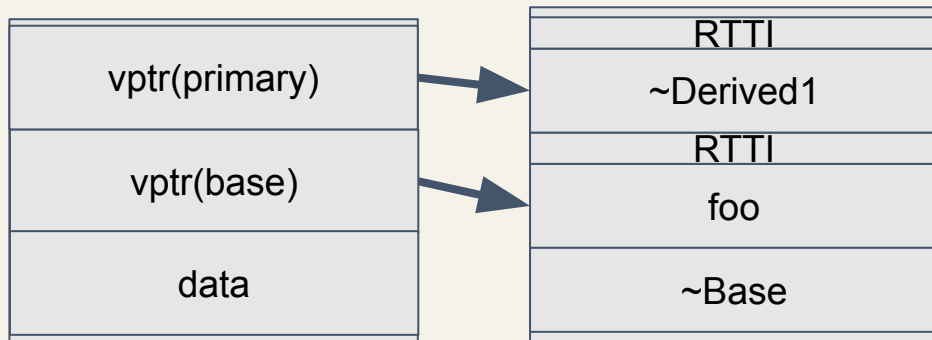


```
struct Derived1 : virtual public Base { ~Derived1() = default; };  
struct Derived2 : virtual public Base { ~Derived2() = default; };
```

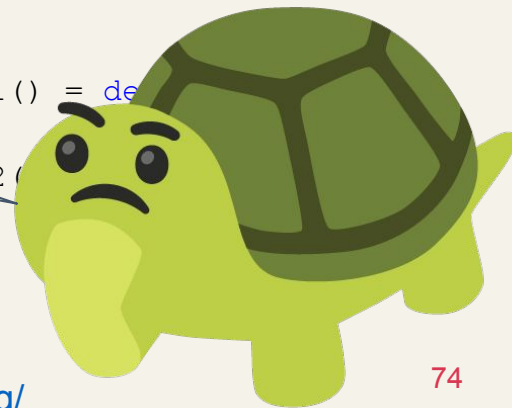
Virtual functions

```
struct Base {  
    virtual void foo() {}  
    virtual ~Base() = default;  
    int data;  
};
```

```
struct Derived1 : virtual public Base {  
    virtual void foo() = default;  
};  
struct Derived2 : virtual public Base {  
    virtual void foo() = default;  
};
```

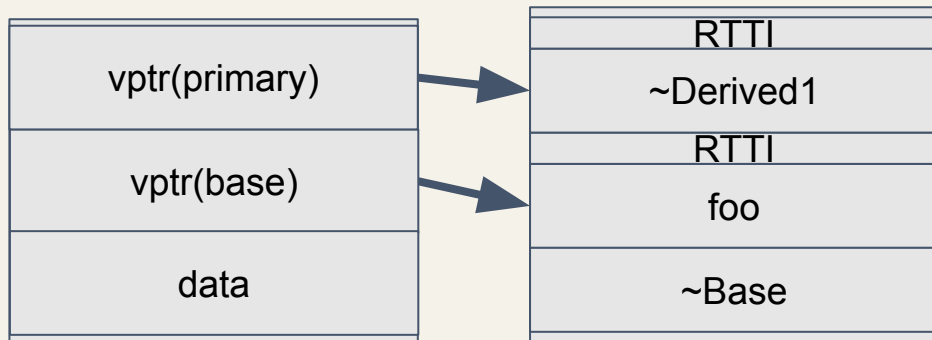


Actually, it has some offset info.

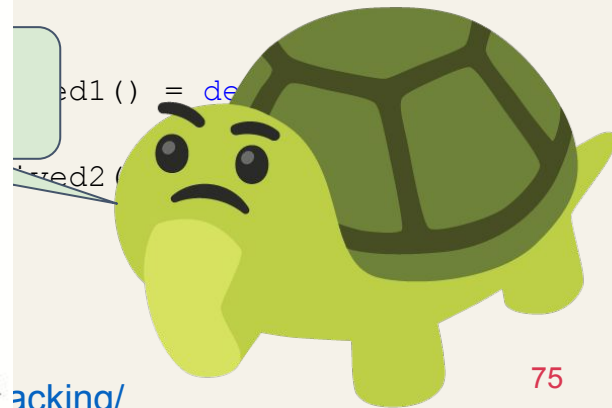


Virtual functions

```
struct Base {  
    virtual void foo() {}  
    virtual ~Base() = default;  
    int data;  
};
```



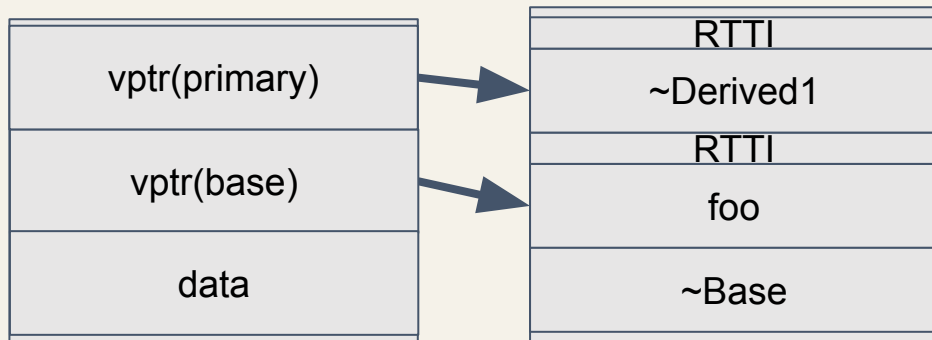
```
444 ~ vtable for Derived1:  
445     .quad    8  
446     .quad    0  
447     .quad    typeinfo for Derived1  
448     .quad    Derived1::~Derived1(.) [complete object destructor]  
449     .quad    Derived1::~Derived1(.) [deleting destructor]  
450     .quad   -8  
451     .quad    0  
452     .quad   -8  
453     .quad    typeinfo for Derived1  
454     .quad    Base::foo(.)  
455     .quad    virtual thunk to Derived1::~Derived1(.) [complete object destructor]  
456     .quad    virtual thunk to Derived1::~Derived1(.) [deleting destructor]
```



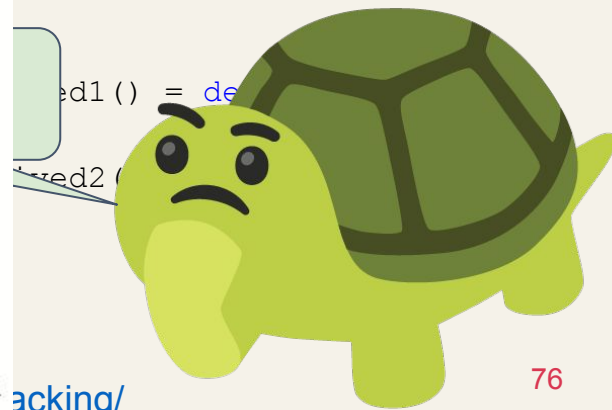
acking/

Virtual functions

```
struct Base {  
    virtual void foo() {}  
    virtual ~Base() = default;  
    int data;  
};
```

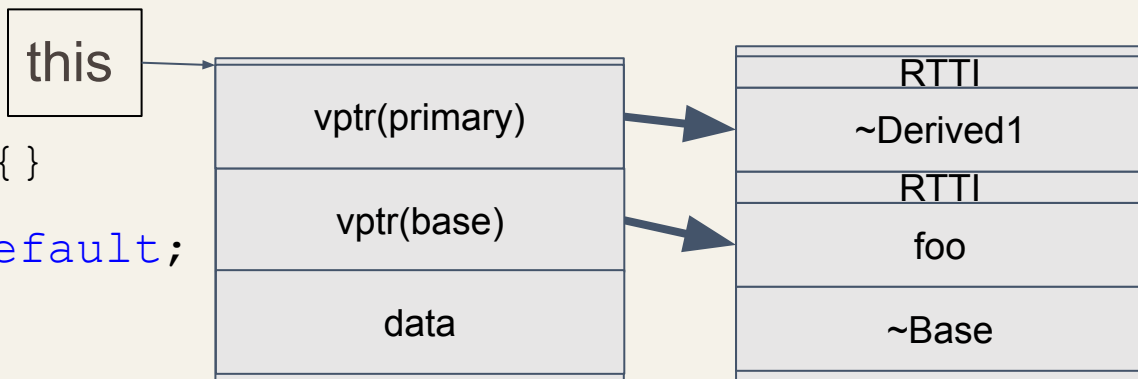


```
444 vtable for Derived1:  
445 .quad 8 ← Offset to Base  
446 .quad 0  
447 .quad typeinfo for Derived1  
448 .quad Derived1::~Derived1(.) [complete object destructor]  
449 .quad Derived1::~Derived1(.) [deleting destructor]  
450 .quad -8  
451 .quad 0  
452 .quad -8  
453 .quad typeinfo for Derived1  
454 .quad Base::foo(.)  
455 .quad virtual thunk to Derived1::~Derived1(.) [complete object destructor]  
456 .quad virtual thunk to Derived1::~Derived1(.) [deleting destructor]
```

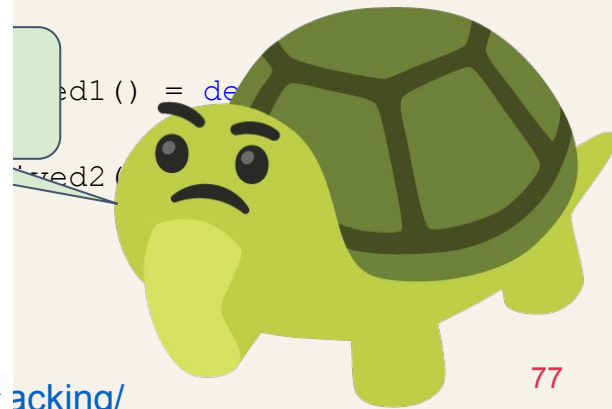


Virtual functions

```
struct Base {  
    virtual void foo() {}  
    virtual ~Base() = default;  
    int data;  
};
```



```
444 vtable for Derived1:  
445 .quad 8 ← Offset to Base  
446 .quad 0  
447 .quad typeinfo for Derived1  
448 .quad Derived1::~Derived1(.) [complete object destructor]  
449 .quad Derived1::~Derived1(.) [deleting destructor]  
450 .quad -8  
451 .quad 0  
452 .quad -8  
453 .quad typeinfo for Derived1  
454 .quad Base::foo(.)  
455 .quad virtual thunk to Derived1::~Derived1(.) [complete object destructor]  
456 .quad virtual thunk to Derived1::~Derived1(.) [deleting destructor]
```

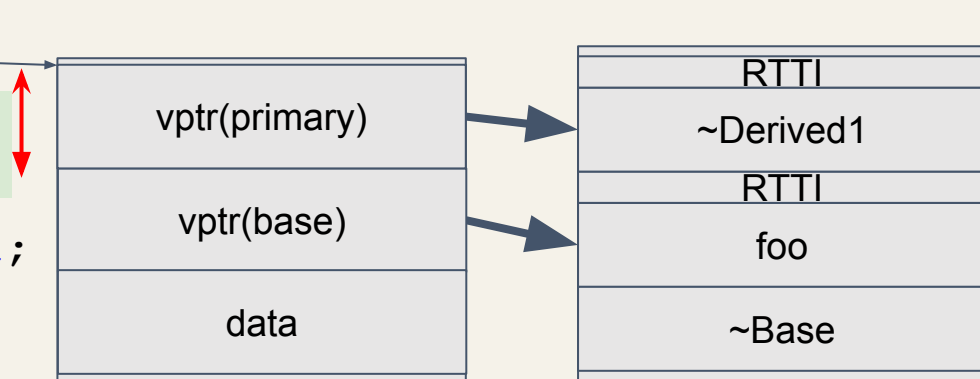


Virtual functions

```
struct Base {  
    virtual void  
    virtual ~Base() = default;  
    int data;  
};
```

this

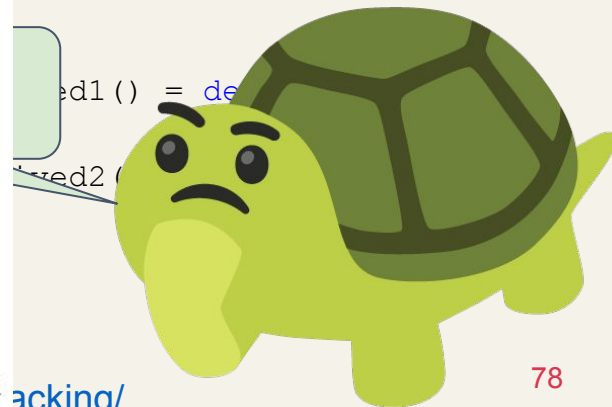
64-bit, 8 byte



444 vtable for Derived1:

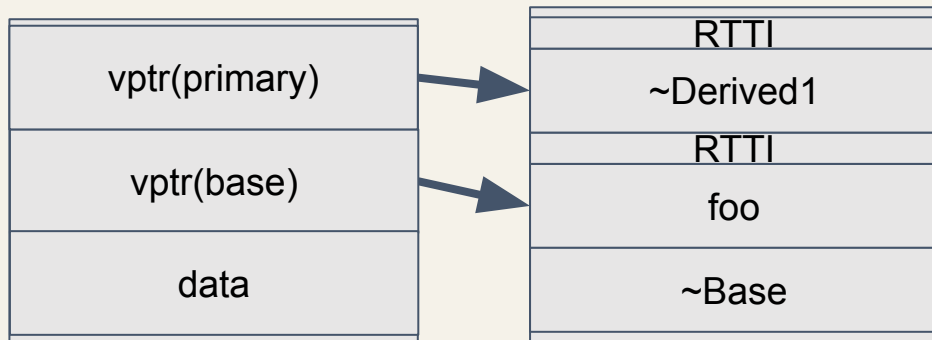
```
445 .quad 8  
446 .quad 0  
447 .quad typeinfo for Derived1  
448 .quad Derived1::~Derived1(.) [complete object destructor]  
449 .quad Derived1::~Derived1(.) [deleting destructor]  
450 .quad -8  
451 .quad 0  
452 .quad -8  
453 .quad typeinfo for Derived1  
454 .quad Base::foo(.)  
455 .quad virtual thunk to Derived1::~Derived1(.) [complete object destructor]  
456 .quad virtual thunk to Derived1::~Derived1(.) [deleting destructor]
```

Offset to Base



Virtual functions

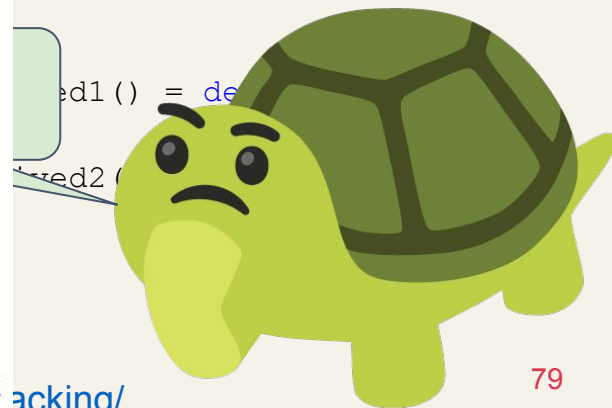
```
struct Base {  
    virtual void foo() {}  
    virtual ~Base() = default;  
    int data;  
};
```



444 vtable for Derived1:

```
445 .quad 8  
446 .quad 0  
447 .quad typeid for Derived1  
448 .quad Derived1::~Derived1() [complete object destructor]  
449 .quad Derived1::~Derived1() [deleting destructor]  
450 .quad -8  
451 .quad 0  
452 .quad -8  
453 .quad typeid for Derived1  
454 .quad Base::foo()  
455 .quad virtual thunk to Derived1::~Derived1() [complete object destructor]  
456 .quad virtual thunk to Derived1::~Derived1() [deleting destructor]
```

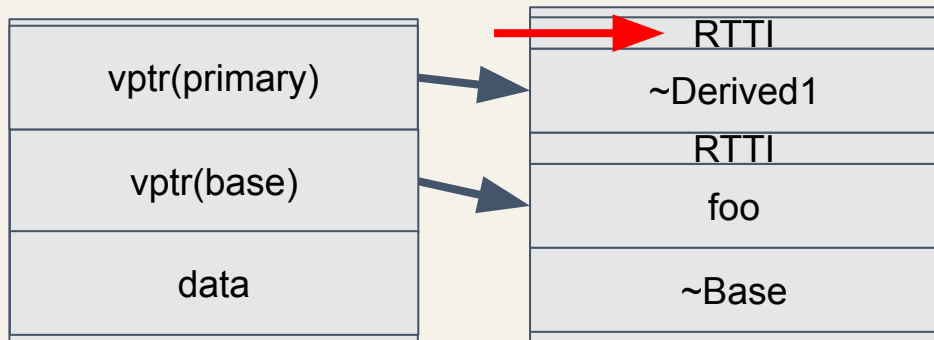
Offset to top



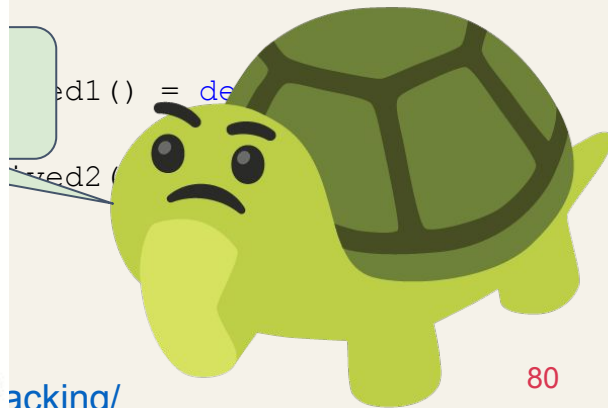
acking/

Virtual functions

```
struct Base {  
    virtual void foo() {}  
    virtual ~Base() = default;  
    int data;  
};
```

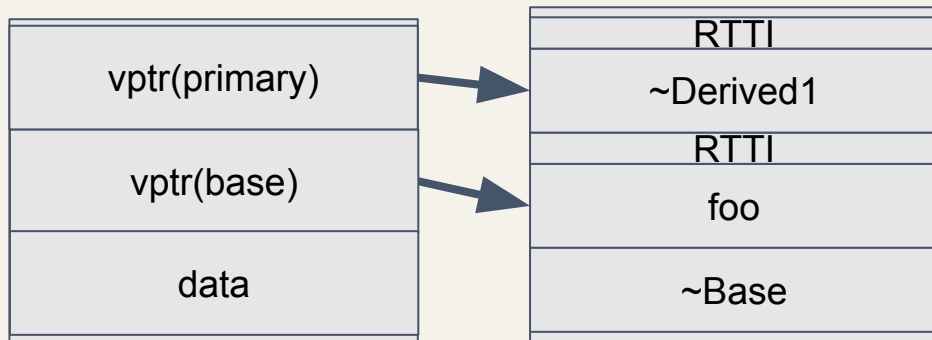


```
444 ~ vtable for Derived1:  
445     .quad 8  
446     .quad 0  
447     .quad typeinfo for Derived1  
448     .quad Derived1::~Derived1(.) [complete object destructor]  
449     .quad Derived1::~Derived1(.) [deleting destructor]  
450     .quad -8  
451     .quad 0  
452     .quad -8  
453     .quad typeinfo for Derived1  
454     .quad Base::foo(.)  
455     .quad virtual thunk to Derived1::~Derived1(.) [complete object destructor]  
456     .quad virtual thunk to Derived1::~Derived1(.) [deleting destructor]
```



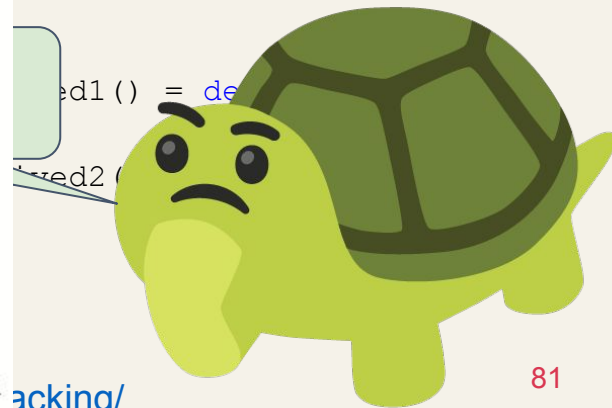
Virtual functions

```
struct Base {  
    virtual void foo() {}  
    virtual ~Base() = default;  
    int data;  
};
```



```
444 ~ vtable for Derived1:  
445     .quad      8  
446     .quad      0  
447     .quad  typeinfo for Derived1  
448     .quad  Derived1::~Derived1(.) [complete object destructor]  
449     .quad  Derived1::~Derived1(.) [deleting destructor]  
450     .quad     -8  
451     .quad      0  
452     .quad     -8  
453     .quad  typeinfo for Derived1  
454     .quad  Base::foo(.)  
455     .quad  virtual thunk to Derived1::~Derived1(.) [complete object destructor]  
456     .quad  virtual thunk to Derived1::~Derived1(.) [deleting destructor]
```

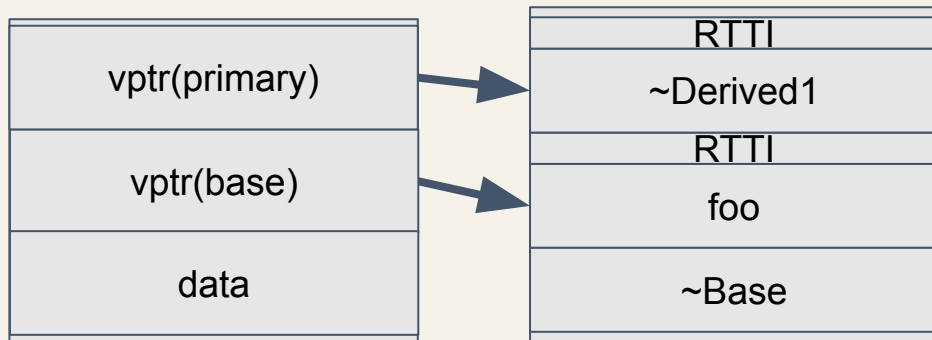
Real dtor



acking/

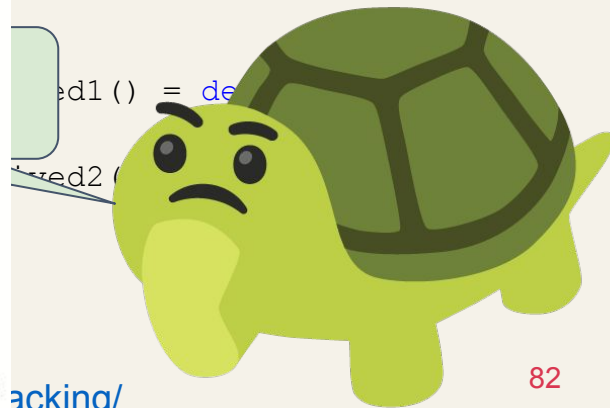
Virtual functions

```
struct Base {  
    virtual void foo() {}  
    virtual ~Base() = default;  
    int data;  
};
```



```
444 ~ vtable for Derived1:  
445     .quad    8  
446     .quad    0  
447     .quad    typeinfo for Derived1  
448     .quad    Derived1::~Derived1(.) [complete object destructor]  
449     .quad    Derived1::~Derived1(.) [deleting destructor]  
450     .quad   -8  
451     .quad    0  
452     .quad   -8  
453     .quad    typeinfo for Derived1  
454     .quad    Base::foo(.)  
455     .quad    virtual thunk to Derived1::~Derived1(.) [complete object destructor]  
456     .quad    virtual thunk to Derived1::~Derived1(.) [deleting destructor]
```

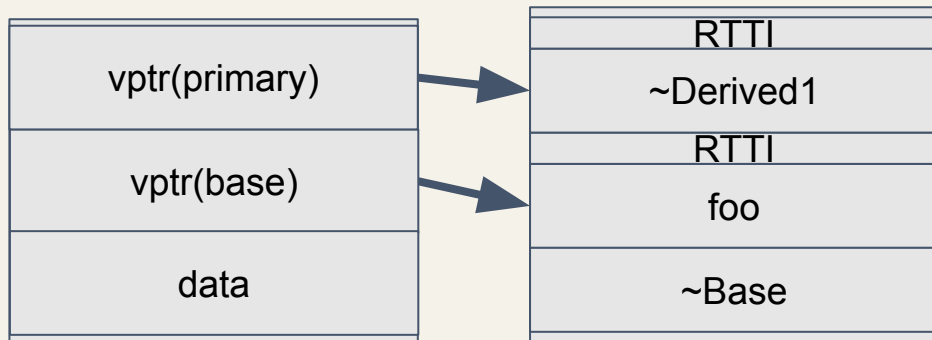
Dtor after dtor



acking/

Virtual functions

```
struct Base {  
    virtual void foo() {}  
    virtual ~Base() = default;  
    int data;  
};
```



```
444 ~ vtable for Derived1:  
445     .quad 8  
446     .quad 0  
447     .quad typeinfo for Derived1  
448     .quad Derived1::~Derived1(.) [complete object destructor]  
449     .quad Derived1::~Derived1(.) [deleting destructor]  
450     .quad -8  
451     .quad 0  
452     .quad -8  
453     .quad typeinfo for Derived1  
454     .quad Base::foo(.)  
455     .quad virtual thunk to Derived1::~Derived1(.) [complete object destructor]  
456     .quad virtual thunk to Derived1::~Derived1(.) [deleting destructor]
```

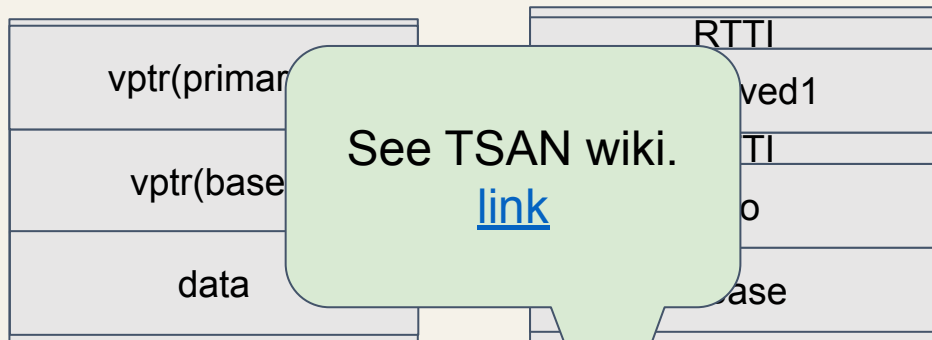
Yes, it's data raceable.

Dtor after dtor



Virtual functions

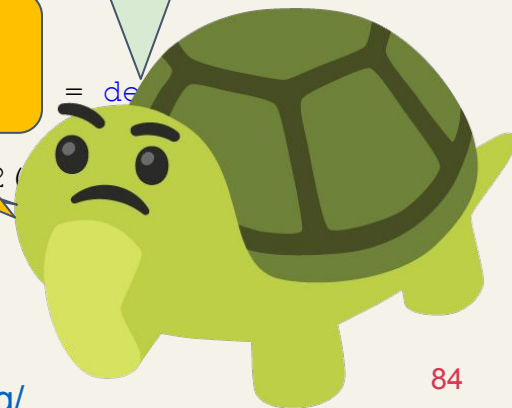
```
struct Base {  
    virtual void foo() {}  
    virtual ~Base() = default;  
    int data;  
};
```



```
444 ~ vtable for Derived1:  
445     .quad 8  
446     .quad 0  
447     .quad typeid for Derived1  
448     .quad Derived1::~Derived1\(\) \[complete object destructor\]  
449     .quad Derived1::~Derived1\(\) \[deleting destructor\]  
450     .quad -8  
451     .quad 0  
452     .quad -8  
453     .quad typeid for Derived1  
454     .quad Base::foo\(\)  
455     .quad virtual thunk to Derived1::~Derived1\(\) \[complete object destructor\]  
456     .quad virtual thunk to Derived1::~Derived1\(\) \[deleting destructor\]
```

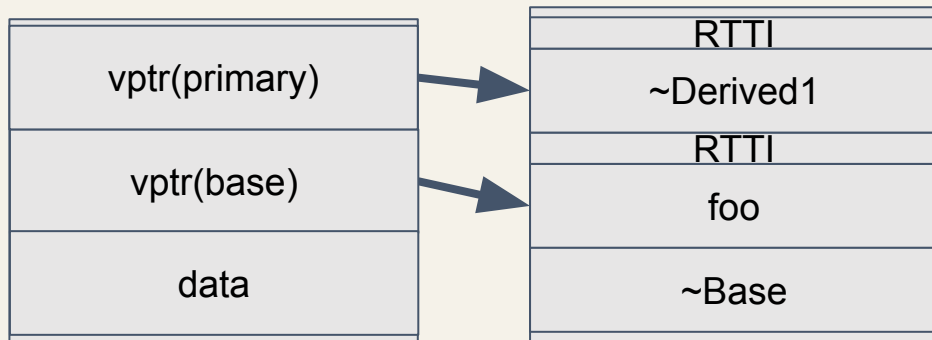
Yes, it's data raceable.

Dtor after dtor



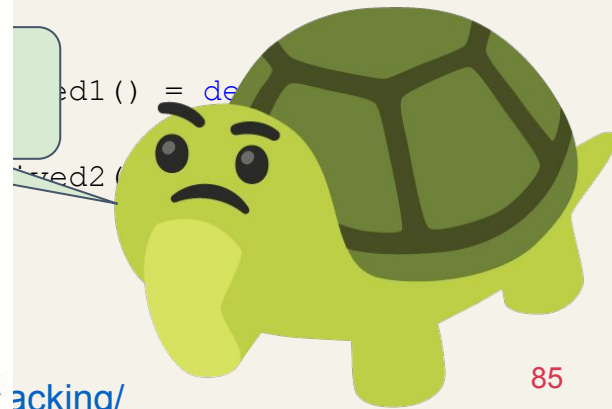
Virtual functions

```
struct Base {  
    virtual void foo() {}  
    virtual ~Base() = default;  
    int data;  
};
```



```
444 ~ vtable for Derived1:  
445     .quad 8  
446     .quad 0  
447     .quad typeinfo for Derived1  
448     .quad Derived1::~Derived1(.) [complete object destructor]  
449     .quad Derived1::~Derived1(.) [deleting destructor]  
450     .quad -8  
451     .quad 0  
452     .quad -8  
453     .quad typeinfo for Derived1  
454     .quad Base::foo(.)  
455     .quad virtual thunk to Derived1::~Derived1(.) [complete object destructor]  
456     .quad virtual thunk to Derived1::~Derived1(.) [deleting destructor]
```

For dynamic_cast...



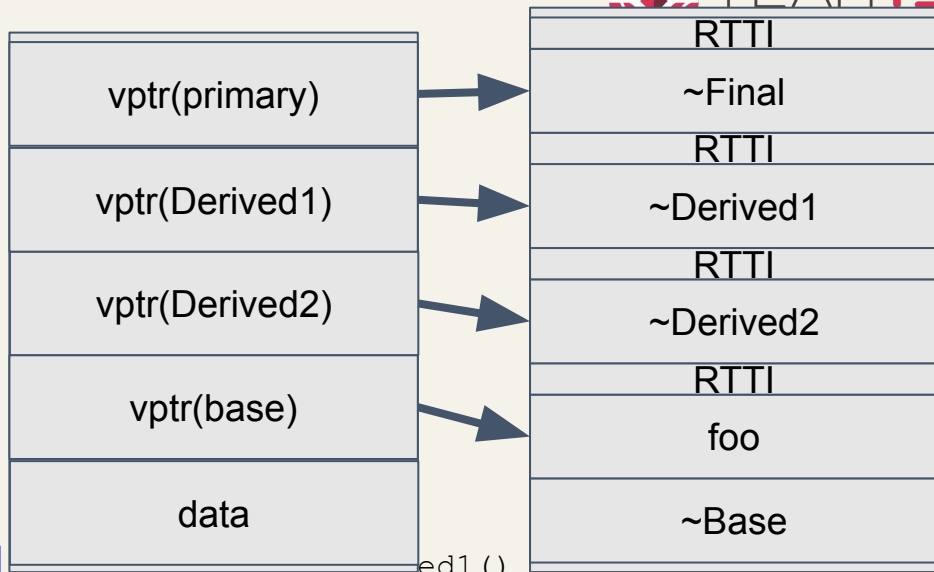
Virtual functions

```
struct Base {  
    virtual void foo() {}  
    virtual ~Base() = default;  
    int data;  
};  
  
struct Derived1 : virtual public Base { ~Derived1() = default; };  
struct Derived2 : virtual public Base { ~Derived2() = default; };  
struct Final    : public Derived1, public Derived2 {  
    ~Final() = default; };
```

Virtual functions

```
struct Base {
    virtual void foo() {}
    virtual ~Base() = default;
    int data;
};
```

```
struct Derived1 : virtual public Base { ~Derived1() = default; };
struct Derived2 : virtual public Base { ~Derived2() = default; };
struct Final    : public Derived1, public Derived2 {
    ~Final() = default; };
```



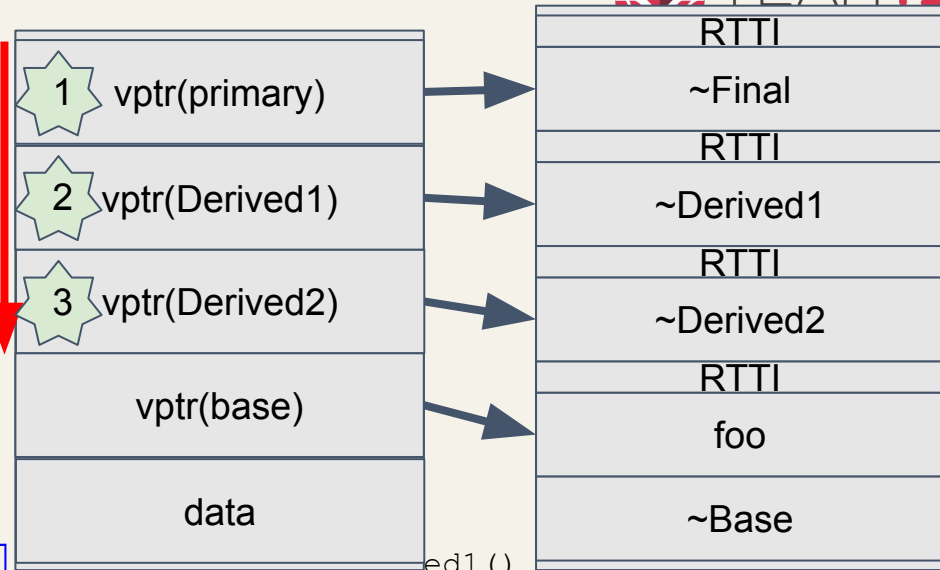
Virtual functions

```
struct Base {
    virtual void foo() {}
    virtual ~Base() = default;
    int data;
};
```

```
struct Derived1 : virtual public Base {
    virtual void foo() { /* Derived1() */ }
```

```
struct Derived2 : virtual public Base {
    virtual ~Derived2() = default; };
```

```
struct Final : public Derived1, public Derived2 {
    ~Final() = default; };
```

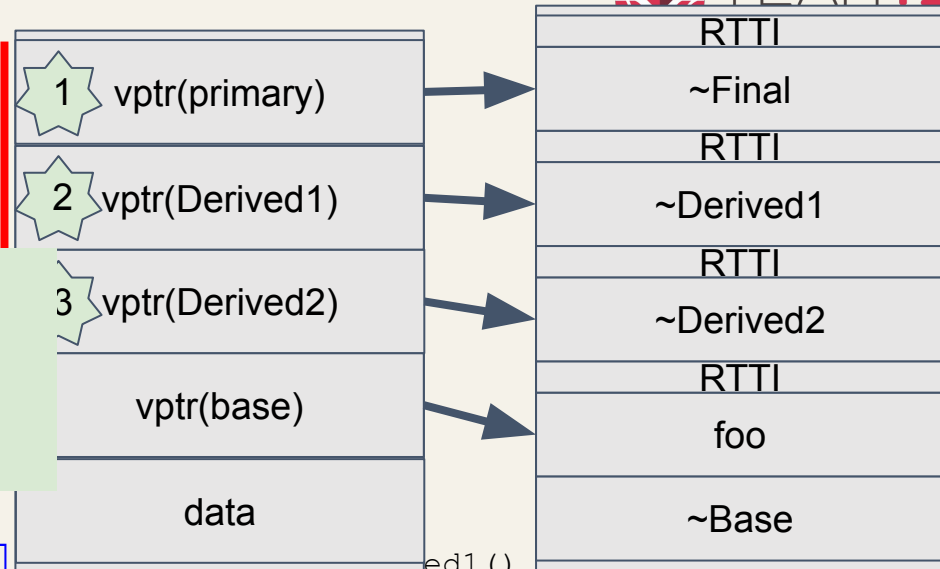


Virtual functions

```
struct Base {
    virtual void foo() {}
    virtual
    int data;
};
```

```
struct Derived1 : virtual public Base { Derived1() {} };
struct Derived2 : virtual public Base { ~Derived2() = default; };
struct Final : public Derived1, public Derived2 {
    ~Final() = default; };
```

Virtual inheritance
would share same
type of data.



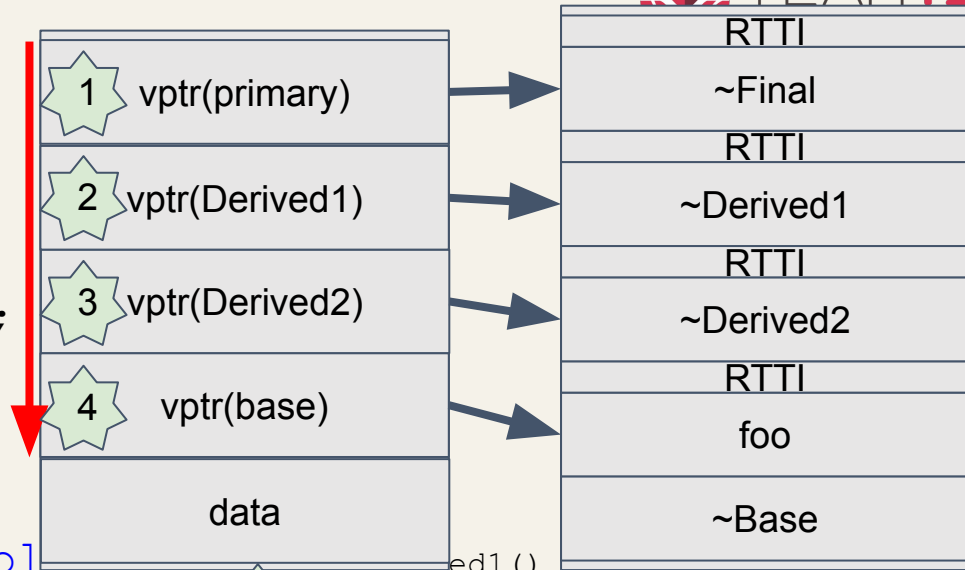
Virtual functions

```
struct Base {
    virtual void foo() {}
    virtual ~Base() = default;
    int data;
};
```

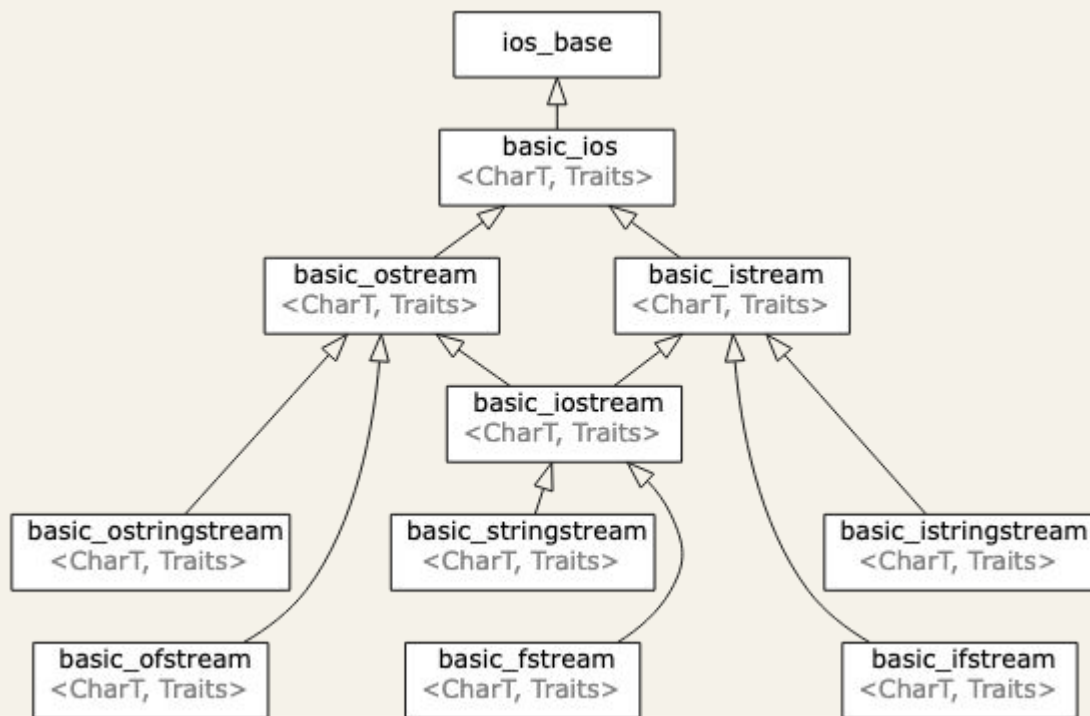
```
struct Derived1 : virtual public Base {
    virtual void foo() {}
    virtual ~Derived1() = default;
};
```

```
struct Derived2 : virtual public Base {
    virtual void foo() {}
    virtual ~Derived2() = default;
};
```

```
struct Final : public Derived1, public Derived2 {
    ~Final() = default;
};
```



Diamond of death



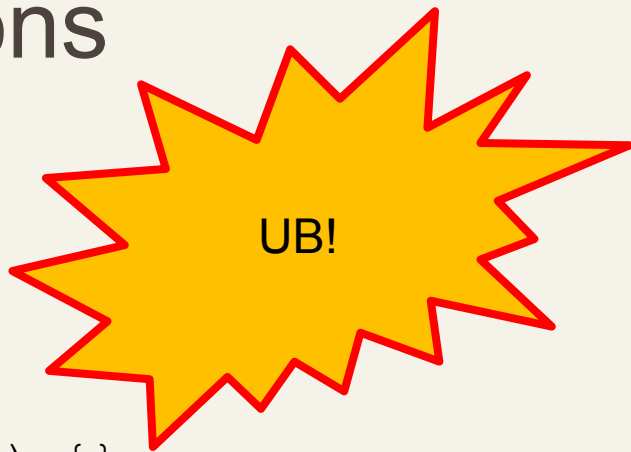
Virtual functions

```
struct Obj {  
    void normal() {}  
    virtual void foo() {}  
virtual Obj = default;  
    int data;  
};
```

What if no dtor?



Virtual functions



```
struct Obj {  
    void normal() {}  
    virtual void foo() {}  
virtual Obj = default;  
    int data;  
};
```

What if no dtor?



Virtual functions



```
struct Obj {  
    void normal() {}  
    virtual void foo() {}  
virtual Obj = default;  
    int data;  
};
```

What if no dtor?



[§ 5.3.5 p5](#) If the object being deleted has incomplete class type at the point of deletion and the complete class has a non-trivial destructor or a deallocation function, the behavior is undefined.

<https://devblogs.microsoft.com/oldnewthing/20200618-00/?p=103874>

<https://gcc.gnu.org/wiki/VerboseDiagnostics#delete-non-virtual-dtor>

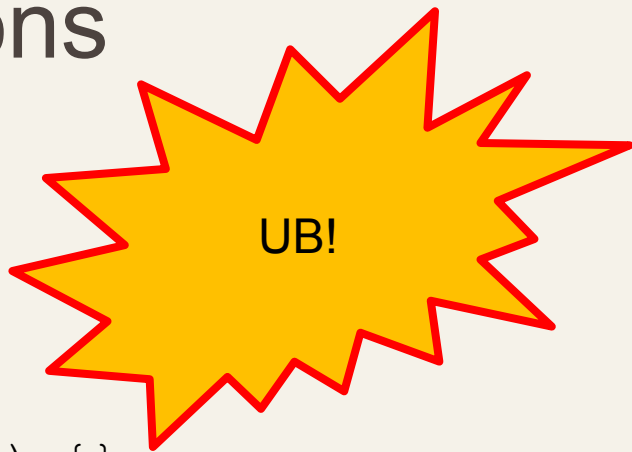
Virtual functions

```
struct Obj {  
    void normal() {}  
    virtual void foo() {}  
    ~Obj = default;  
    int data;  
};
```

What if non-virtual?



Virtual functions



```
struct Obj {  
    void normal() {}  
    virtual void foo() {}  
    ~Obj = default;  
    int data;  
};
```

What if non-virtual?



[§ 5.3.5 p5](#) If the object being deleted has incomplete class type at the point of deletion and the complete class has a non-trivial destructor or a deallocation function, the behavior is undefined. <https://godbolt.org/z/Gq6P5TWK3>
<https://devblogs.microsoft.com/oldnewthing/20200618-00/?p=103874>
<https://gcc.gnu.org/wiki/VerboseDiagnostics#delete-non-virtual-dtor>

Calling based virtual

```
struct Base {
    void goo() {puts(__PRETTY_FUNCTION__);}
    virtual void foo() {puts(__PRETTY_FUNCTION__);}
    void use_thread() {
        std::thread t(&Base::foo, this);
        t.join();
    }
    virtual ~Base() = default;
};

struct Derived : public Base {
    void goo() {puts(__PRETTY_FUNCTION__);}
    virtual void foo() override {puts(__PRETTY_FUNCTION__);}
    virtual ~Derived() = default;
};
```

Calling based virtual

```
struct Base {
    void goo() {puts(__PRETTY_FUNCTION__);}
    virtual void foo() {puts(__PRETTY_FUNCTION__);}
    void use_thread() {
        std::thread t(&Base::foo, this);
        t.join();
    }
    virtual ~Base() = default;
};

int main() {
    Derived d;
    d.use_thread();
}

struct Derived : public Base {
    void goo() {puts(__PRETTY_FUNCTION__);}
    virtual void foo() override {puts(__PRETTY_FUNCTION__);}
    virtual ~Derived() = default;
};
```

Calling based virtual

```
struct Base {
    void goo() {puts(__PRETTY_FUNCTION__);}
    virtual void foo() {puts(__PRETTY_FUNCTION__);}
    void use_thread() {
        std::thread t(&Base::foo, this);
        t.join();
    }
    virtual ~Base() = default;
};

int main() {
    Derived d;
    d.use_thread();
}

struct Derived : public Base {
    void goo() {puts(__PRETTY_FUNCTION__);}
    virtual void foo() override {puts(__PRETTY_FUNCTION__);}
    virtual ~Derived() = default;
};
```

Calling based virtual

```
struct Base {
    void goo() {puts(__PRETTY_FUNCTION__);}
    virtual void foo() {puts(__PRETTY_FUNCTION__);}
    void use_thread() {
        std::thread t(&Base::foo, this);
        t.join();
    }
    virtual ~Base() = default;
};

int main() {
    Derived d;
    d.use_thread();
}

struct Derived : public Base {
    void goo() {puts(__PRETTY_FUNCTION__);}
    virtual void foo() override {puts(__PRETTY_FUNCTION__);}
    virtual ~Derived() = default;
};
```

Calling based virtual

```
struct Base {
    void goo() {puts(__PRETTY_FUNCTION__);}
    virtual void foo() {puts(__PRETTY_FUNCTION__);}
    void use_thread() {
        std::thread t(&Base::foo, this);
        t.join();
    }
    virtual ~Base() = default;
};

int main() {
    Derived d;
    d.use_thread();
}

struct Derived : public Base {
    void goo() {puts(__PRETTY_FUNCTION__);}
    virtual void foo() override {puts(__PRETTY_FUNCTION__);}
    virtual ~Derived() = default;
};
```

Calling based virtual

```
struct Base {  
    void goo() {puts(__PRETTY_FUNCTION__);}  
    virtual void foo() {puts(__PRETTY_FUNCTION__);}  
    void use_thread() {  
        std::thread t(&Base::foo, this);  
        t.join();  
    }  
    virtual ~Base() = default;  
};
```

```
struct Derived : public Base {  
    void goo() {puts(__PRETTY_FUNCTION__);}  
    virtual void foo() override {puts(__PRETTY_FUNCTION__);}  
    virtual ~Derived() = default;  
};
```

```
int main() {  
    Derived d;  
    d.use_thread();  
}
```

```
Execution ended, compiler returned: 0  
Program returned: 0  
virtual void Derived::foo()
```

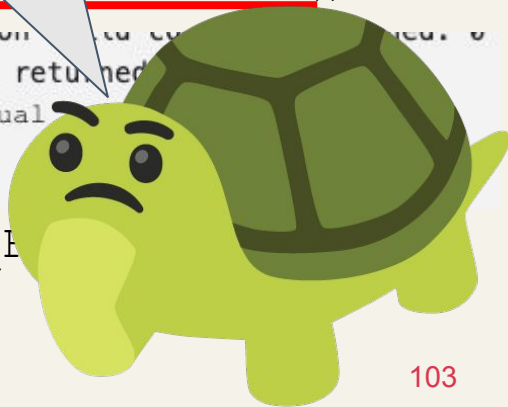
Calling based virtual

```
struct Base {  
    void goo() {puts(__PRETTY_FUNCTION__);}  
    virtual void foo() {puts(__PRETTY_FUNCTION__);}  
    void use_thread() {  
        std::thread t(&Base::foo, this);  
        t.join();  
    }  
    virtual ~Base() = default;  
};
```

```
struct Derived : public Base {  
    void goo() {puts(__PRETTY_FUNCTION__);}  
    virtual void foo() override {puts(__PRETTY_FUNCTION__);}  
    virtual ~Derived() = default;  
};
```

[§11.7.3 p10](#) [Note 3: The interpretation of the call of a virtual function depends on the type of the object for which it is called (the dynamic type), whereas the interpretation of a call of a non-virtual member function depends only on the type of the pointer or reference denoting that object (the static type) ([expr.call]). — end note]

...
 void d;
 ...
 use_thread();
 ...
 Program returned
 virtual



Calling based virtual

```
struct Base {  
    void goo() {puts(__PRETTY_FUNCTION__);}  
    virtual void foo() {puts(__PRETTY_FUNCTION__);}  
    void use_thread() {  
        std::thread t(&Base::foo, this);  
        t.join();  
    }  
    virtual ~Base() = default;  
};
```

```
struct Derived : public Base {  
    void goo() {puts(__PRETTY_FUNCTION__);}  
    virtual void foo() override {puts(__PRETTY_FUNCTION__);}  
    virtual ~Derived() = default;  
};
```

[§11.7.3 p10](#) [Note 3: The interpretation of the call of a virtual function depends on the type of the object for which it is called (the dynamic type), whereas the interpretation of a call of a non-virtual member function depends only on the type of the pointer or reference denoting that object (the static type) ([expr.call]). — end note]

...
 void foo() {
 ...
 d.use_thread();
 }
...
Program returned
virtual



Disallow heap allocation

```
struct O {  
    void *operator new(size_t)  
        = delete;  
    void operator delete(void*)  
        = delete;  
};  
  
int main() {  
    auto op =  
        std::make_unique<O>();  
}
```

```
std::make_unique(_Args&& ...) [with _Tp = O; _Args = {}; __detail::__unique_ptr_t<_Tp> = __detail::__unique_ptr_t<O>]:  
<source>:11:34:   required from here  
11 |     auto op = std::make_unique<O>();  
    |               ~~~~~~  
/opt/compiler-explorer/gcc-trunk-20231201/include/c++/14.0.0/bits/unique_ptr.h:1076:30: error: use of deleted function 'static void* O::operator  
new(size_t)'  
1076 |     { return unique_ptr<_Tp>(new _Tp(std::forward<_Args>(__args)...)); }  
    |                               ~~~~~~  
<source>:6:11: note: declared here  
6 |     void *operator new(size_t) = delete;  
    |     ~~~~~~  
/opt/compiler-explorer/gcc-trunk-20231201/include/c++/14.0.0/bits/unique_ptr.h: In instantiation of 'void std::default_delete<_Tp>::operator()  
(_Tp*) const [with _Tp = O]':  
/opt/compiler-explorer/gcc-trunk-20231201/include/c++/14.0.0/bits/unique_ptr.h:398:17:   required from 'std::unique_ptr<_Tp, _Dp>::~unique_ptr()  
[with _Tp = O; _Dp = std::default_delete<O>]'  
<source>:6:11: note:   398 |         get_deleter()(std::move(__ptr));  
<source>:6:11: note:   ~~~~~~  
<source>:11:34:   required from here  
<source>:6:11: note:   11 |     auto op = std::make_unique<O>();  
<source>:6:11: note:   ~~~~~~  
/opt/compiler-explorer/gcc-trunk-20231201/include/c++/14.0.0/bits/unique_ptr.h:93:9: error: use of deleted function 'static void O::operator  
delete(void*)'  
93 |         delete __ptr;  
    |         ~~~~~~  
<source>:7:10: note: declared here  
7 |     void operator delete(void *) = delete;  
    |     ~~~~~~  
Compiler returned: 1
```

Disallow heap allocation

```
struct O {  
    void *operator new(size_t)  
        = delete;  
    void operator delete(void*)  
        = delete;  
};  
  
int main() {  
    auto op =  
        std::make_unique<O>();  
}
```

```
std::make_unique(_Args&& ...) [with _Tp = O; _Args = {}; __detail::__unique_ptr_t<_Tp> = __detail::__unique_ptr_t<O>]:  
<source>:11:34:   required from here  
11 |     auto op = std::make_unique<O>();  
    |               ~~~~~~  
/opt/compiler-explorer/gcc-trunk-20231201/include/c++/14.0.0/bits/unique_ptr.h:1076:30: error: use of deleted function 'static void* O::operator  
new(size_t)'  
1076 |     { return unique_ptr<_Tp>(new _Tp(std::forward<_Args>(_args)...)); }  
    |                               ~~~~~~  
<source>:6:11: note: declared here  
6 |     void *operator new(size_t) = delete;  
    |     ~~~~~~  
/opt/compiler-explorer/gcc-trunk-20231201/include/c++/14.0.0/bits/unique_ptr.h: In instantiation of 'void std::default_delete<_Tp>::operator()  
(_Tp*) const [with _Tp = O]':  
/opt/compiler-explorer/gcc-trunk-20231201/include/c++/14.0.0/bits/unique_ptr.h:398:17:   required from 'std::unique_ptr<_Tp, _Dp>::~unique_ptr()  
[with _Tp = O; _Dp = std::default_delete<O>]'  
<source>:6:11: note:   398 |         get_deleter()(std::move(__ptr));  
<source>:6:11: note:   |  
<source>:11:34:   required from here  
<source>:6:11: note:   11 |     auto op = std::make_unique<O>();  
<source>:6:11: note:   |  
/opt/compiler-explorer/gcc-trunk-20231201/include/c++/14.0.0/bits/unique_ptr.h:93:9: error: use of deleted function 'static void O::operator  
delete(void*)'  
93 |         delete __ptr;  
    |         ~~~~~~  
<source>:7:10: note: declared here  
7 |     void operator delete(void *) = delete;  
    |     ~~~~~~  
Compiler returned: 1
```

Overload the new/delete
operator.

Disallow heap allocation

```
struct O {  
    void *operator new(size_t)  
        = delete;  
    void operator delete(void*)  
        = delete;  
};  
  
int main() {  
    auto op =  
        std::make_unique<O>();  
}
```

Use malloc to bypass.

```
#define define_obj_on_heap(type) ({  
    void* ptr = std::malloc(sizeof(type));  
    reinterpret_cast<type*>(ptr);  
})
```

```
std::make_unique<Args66 ...> [with _Tp = O; _Args = {}; __detail::__unique_ptr_t<_Tp> = __detail::__unique_ptr_t<O>]':  
<source>:11:34:   required from here  
11 |     auto op = std::make_unique<O>();  
    |               ~~~~~~  
/opt/compiler-explorer/gcc-trunk-20231201/include/c++/14.0.0/bits/unique_ptr.h:1076:30: error: use of deleted function 'static void* O::operator  
new(size_t)'  
1076 |     { return unique_ptr<_Tp>(new _Tp(std::forward<Args>(__args)...)); }  
    |                               ~~~~~~  
<source>:6:11: note: declared here  
6 |     void *operator new(size_t) = delete;  
    |     ~~~~~~  
/opt/compiler-explorer/gcc-trunk-20231201/include/c++/14.0.0/bits/unique_ptr.h: In instantiation of 'void std::default_delete<_Tp>::operator()  
(_Tp*) const [with _Tp = O]':  
/opt/compiler-explorer/gcc-trunk-20231201/include/c++/14.0.0/bits/unique_ptr.h:398:17:   required from 'std::unique_ptr<_Tp, _Dp>::~unique_ptr()  
[with _Tp = O; _Dp = std::default_delete<O>]'  
<source>:6:11: note:   398 |     get_deleter()(std::move(__ptr));  
<source>:6:11: note:   |  
<source>:11:34:   required from here  
<source>:6:11: note:   11 |     auto op = std::make_unique<O>();  
<source>:6:11: note:   |  
/opt/compiler-explorer/gcc-trunk-20231201/include/c++/14.0.0/bits/unique_ptr.h:93:9: error: use of deleted function 'static void O::operator  
delete(void*)'  
93 |     delete __ptr;  
    |     ~~~~~~  
<source>:7:10: note: declared here  
7 |     void operator delete(void *) = delete;  
    |     ~~~~~~  
Compiler returned: 1
```

Disallow stack allocation

```
struct O {  
    ~O() = delete;  
};
```

```
int main() {  
    O o;  
}
```

```
<source>: In function 'int main()':  
<source>:6:7: error: use of deleted function 'O::~~O()'  
    6 |     O o;  
      |     ^  
<source>:2:1: note: declared here  
    2 | ~O() = delete;  
      | ^  
Compiler returned: 1
```

Disallow stack allocation

```
struct O {  
    ~O() = delete;  
};
```

```
int main() {  
    O o;  
}
```

Dtor guaranteed
to be invoked.

```
<source>: In function 'int main()':  
<source>:6:7: error: use of deleted function 'O::~~O()'  
    6 |     O o;  
      |     ^  
<source>:2:1: note: declared here  
    2 | ~O() = delete;  
      | ^  
Compiler returned: 1
```

Disallow stack allocation

```
struct O {  
    ~O() = delete;  
};
```

```
int main() {  
    O o;  
}
```

```
<source>: In function 'int main()':  
<source>:6:7: error: use of deleted function 'O::~~O()'  
    6 |     O o;  
      |     ^  
<source>:2:1: note: declared here  
    2 | ~O() = delete;  
      | ^  
Compiler returned: 1
```

```
#define define_obj_on_stack(type) ({ \\\n    char obj_intl__[sizeof(type)]; \\\n    reinterpret_cast<type*>(obj_intl__); \\\n    })
```

Use pointer to bypass.

```
int main() {  
    auto op = define_obj_on_stack(O);  
}
```

Conceptually yes,
but actually no.

Extra: sequence before

```
struct O {};  
  
O &f() {  
    O o;  
    puts(__PRETTY_FUNCTION__);  
    return o;  
}
```

```
O &g() {  
    O o;  
    puts(__PRETTY_FUNCTION__);  
    return o;  
}
```

```
int main() {  
    f() = g();  
    f().operator=(g());  
}
```

Program returned:

O& g()

O& f()

O& g()

O& f()

gcc

Program returned: 0

O &g()

O &f()

O &f()

O &g()

clang

Extra: sequence before

```
struct O {};
```

```
O &f() {  
    O o;  
    puts(__PRETTY_FUNCTION__);  
    return o;  
}
```

```
O &g() {  
    O o;  
    puts(__PRETTY_FUNCTION__);  
    return o;  
}
```

```
int main() {  
    f() = g();  
    f().operator=(g());  
}
```

§8.18 p1[expr.ass] The right operand is sequenced before the left operand.

§16.3.1.2 p2 [over.match.oper] ...function-call notation...the operands are sequenced in the order prescribed for the built-in operator.

Program returned:

O& g()

O& f()

O& g()

O& f()

gcc

Program returned: 0

O &g()

O &f()

O &f()

O &g()

clang

Extra: sequence before

```
struct O {};
```

```
O &f() {  
    O o;  
    puts(__PRETTY_FUNCTION__);  
    return o;  
}
```

```
O &g() {  
    O o;  
    puts(__PRETTY_FUNCTION__);  
    return o;  
}
```

```
int main() {  
    f() = g();  
    f().operator=(g());  
}
```

§8.18 p1[expr.ass] The right operand is sequenced before the left operand.

§16.3.1.2 p2 [over.match.oper] ...function-call notation...the operands are sequenced in the order prescribed for the built-in operator.

Program returned:

O& g()

O& f()

O& g()

O& f()

gcc

Program returned: 0

O &g()

O &f()

O &f()

O &g()

clang

Extra: sequence before

```
struct O {};
```

```
O &f() {  
    O o;  
    puts(__PRETTY_FUNCTION__);  
    return o;  
}
```

```
O &g() {  
    O o;  
    puts(__PRETTY_FUNCTION__);  
    return o;  
}
```

```
int main() {  
    f() = g();  
    f().operator=(g());  
}
```

§8.18 p1[expr.ass] The right operand is sequenced before the left operand.

§16.3.1.2 p2 [over.match.oper] ...function-call notation...the operands are sequenced in the order prescribed for the built-in operator.

Program returned:

O& g()

O& f()

O& g()

O& f()

gcc

Program returned: 0

O &g()

O &f()

O &f()

O &g()

clang

Extra: sequence before

```
struct O {};  
  
O &f() {  
    O o;  
    puts(__PRETTY_FUNCTION__);  
    return o;  
}  
  
O &g() {  
    O o;  
    puts(__PRETTY_FUNCTION__);  
    return o;  
}
```

```
int main() {  
    f() = g();  
    f().operator=(g());  
}
```

So, clang's one is a bug.

Program returned:

O& g()

O& f()

O& g()

O& f()

gcc

Program returned: 0

O &g()

O &f()

O &f()

O &g()

clang

Thank you!

scc@teamt5.org

