System Programming with C++

Pointers revisited, lvalue reference, const

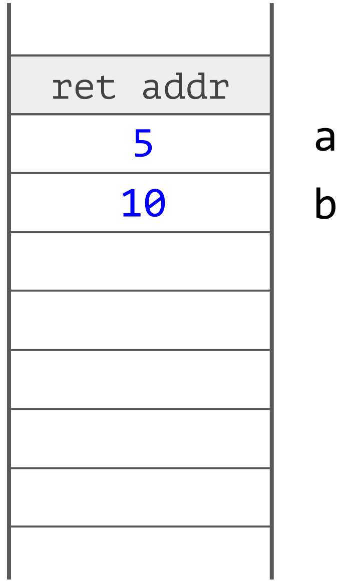
Pointers

Pointers

```
int a = 5;
int b = 10;
```

Pointers

```
int a = 5;
int b = 10;
```



ret addr

5 a

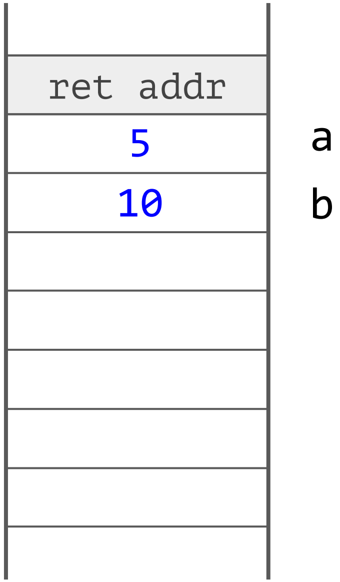
10 b

Pointers

```
int a = 5;
int b = 10;

int* pa = &a;
```

pointer
to int

taking an
address of a



ret addr

5 a

10 b

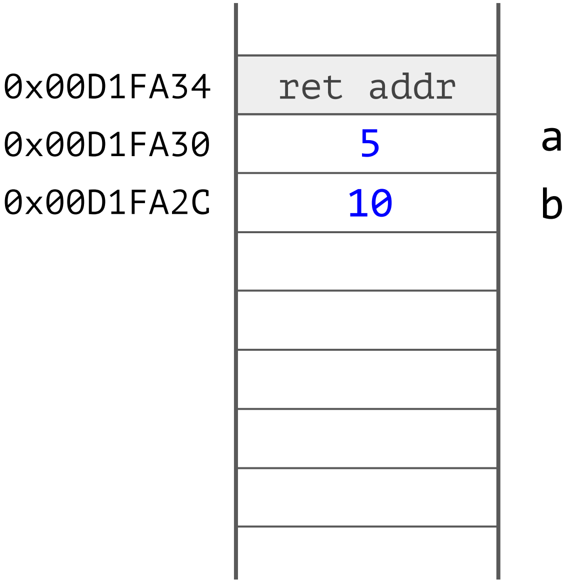
let it be 32-bit architecture in this example, just to keep it simple

Pointers

```
int a = 5;
int b = 10;

int* pa = &a;
```

pointer to int

taking an address of a



0x00D1FA34	ret addr	
0x00D1FA30	5	a
0x00D1FA2C	10	b

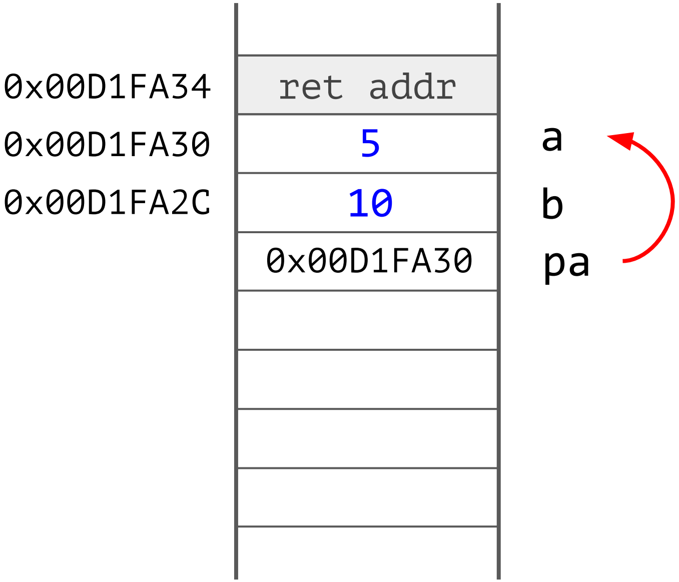
let it be 32-bit architecture in this example, just to keep it simple

Pointers

```
int a = 5;
int b = 10;

int* pa = &a;
```

pointer
to int

taking an
address of a

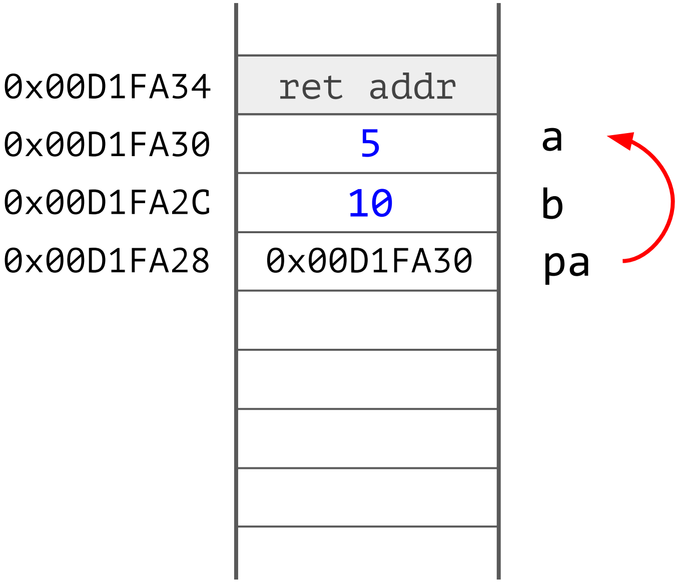


0x00D1FA34	ret addr	
0x00D1FA30	5	a
0x00D1FA2C	10	b
	0x00D1FA30	pa

let it be 32-bit architecture in this example, just to keep it simple

Pointers

```
int a = 5;
int b = 10;

int* pa = &a;
```



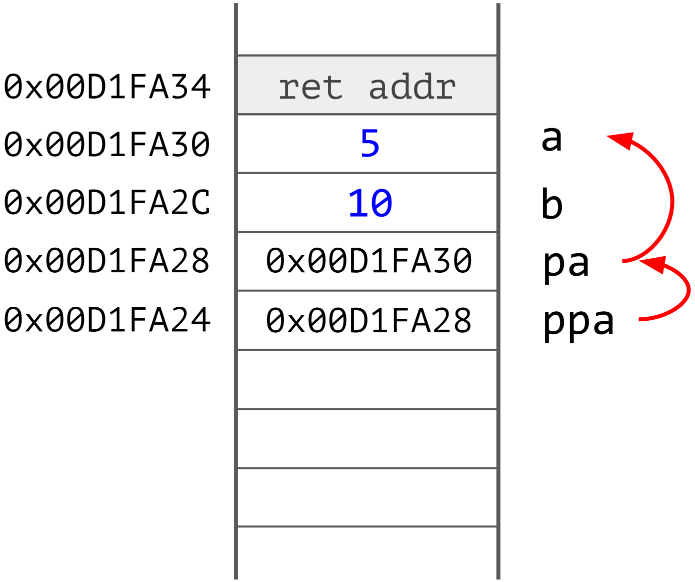
0x00D1FA34	ret addr	
0x00D1FA30	5	a
0x00D1FA2C	10	b
0x00D1FA28	0x00D1FA30	pa

pointers themselves are values (in some memory, with addresses, etc.)

let it be 32-bit architecture in
this example, just to keep it simple

Pointers

```c
int a = 5;
int b = 10;

int* pa = &a;
int** ppa = &pa;
```

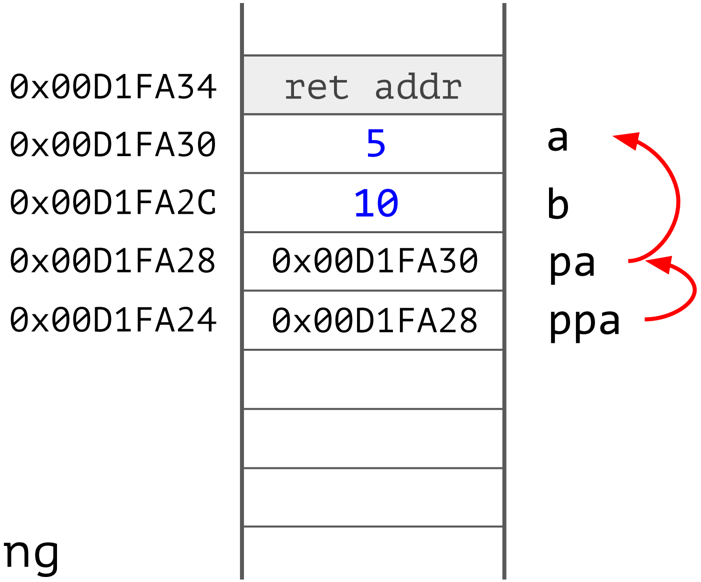


0x00D1FA34	ret addr	
0x00D1FA30	5	a
0x00D1FA2C	10	b
0x00D1FA28	0x00D1FA30	pa
0x00D1FA24	0x00D1FA28	ppa

let it be 32-bit architecture in
this example, just to keep it simple

Pointers

```
int a = 5;
int b = 10;

int* pa = &a;
int** ppa = &pa;

int c = *pa;
```



0x00D1FA34	ret addr	
0x00D1FA30	5	a
0x00D1FA2C	10	b
0x00D1FA28	0x00D1FA30	pa
0x00D1FA24	0x00D1FA28	ppa

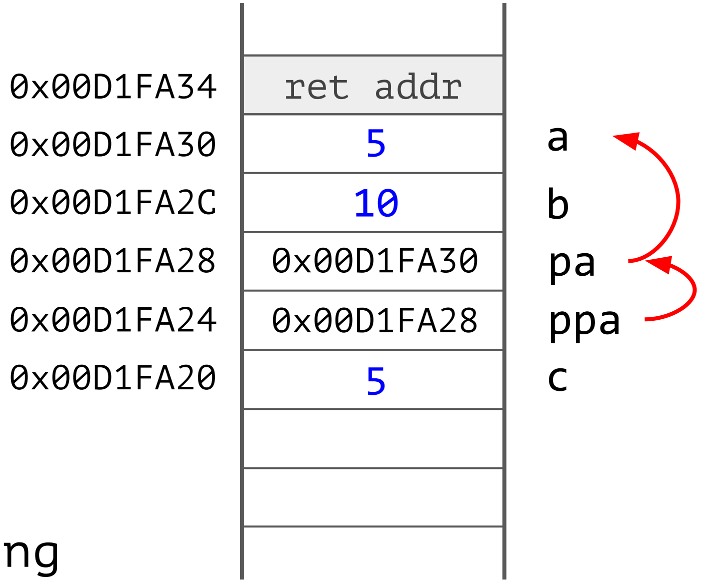
dereferencing
of a pointer

Pointers

let it be 32-bit architecture in this example, just to keep it simple

```
int a = 5;
int b = 10;

int* pa = &a;
int** ppa = &pa;

int c = *pa;
```

dereferencing of a pointer



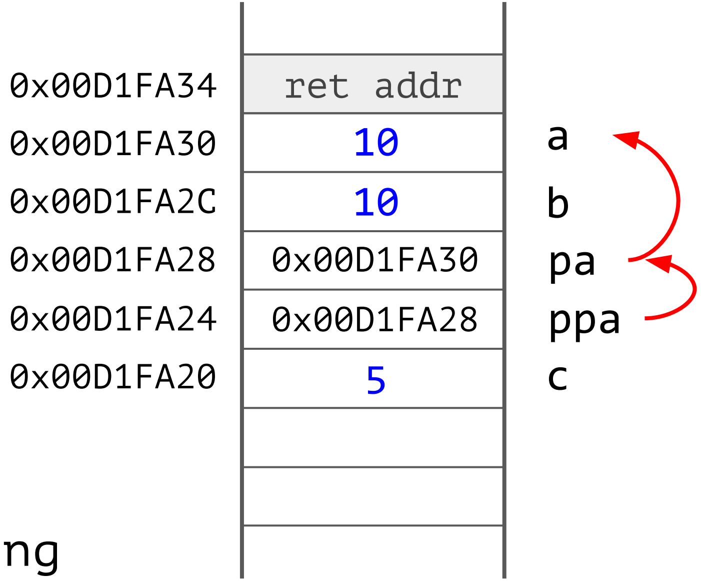
0x00D1FA34	ret addr	
0x00D1FA30	5	a
0x00D1FA2C	10	b
0x00D1FA28	0x00D1FA30	pa
0x00D1FA24	0x00D1FA28	ppa
0x00D1FA20	5	c

let it be 32-bit architecture in this example, just to keep it simple

Pointers

```
int a = 5;
int b = 10;

int* pa = &a;
int** ppa = &pa;

int c = *pa;
*pa = b;
```

dereferencing of a pointer

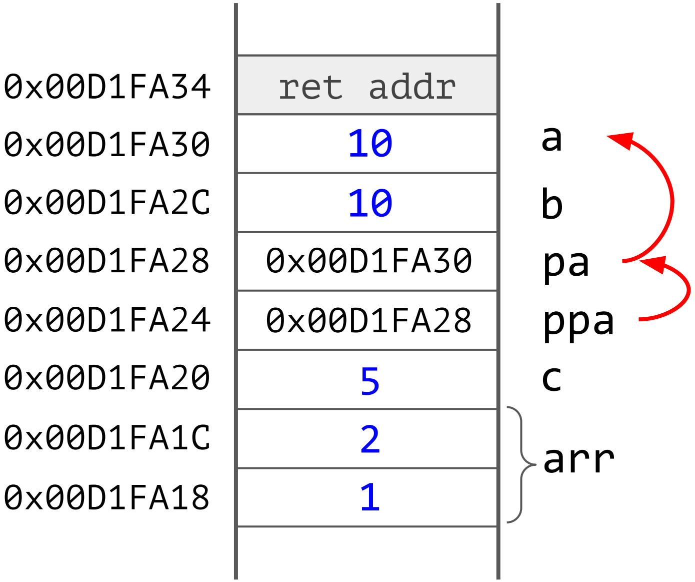


0x00D1FA34	ret addr	
0x00D1FA30	10	a
0x00D1FA2C	10	b
0x00D1FA28	0x00D1FA30	pa
0x00D1FA24	0x00D1FA28	ppa
0x00D1FA20	5	c

let it be 32-bit architecture in this example, just to keep it simple

Pointers

```
int a = 5;
int b = 10;

int* pa = &a;
int** ppa = &pa;
...

int arr[2] = {1, 2};
```

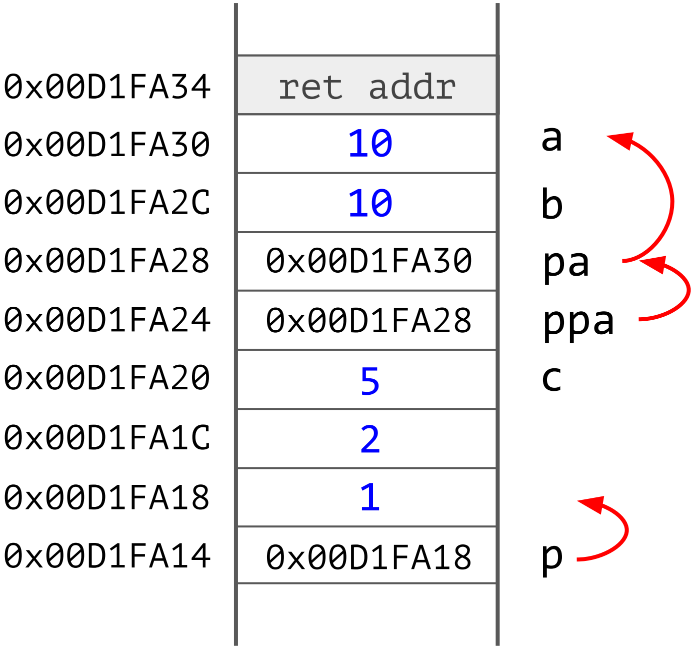


0x00D1FA34	ret addr	
0x00D1FA30	10	a
0x00D1FA2C	10	b
0x00D1FA28	0x00D1FA30	pa
0x00D1FA24	0x00D1FA28	ppa
0x00D1FA20	5	c
0x00D1FA1C	2	arr
0x00D1FA18	1	

let it be 32-bit architecture in this example, just to keep it simple

Pointers

```
int a = 5;
int b = 10;

int* pa = &a;
int** ppa = &pa;
...

int arr[2] = {1, 2};
int* p = arr;
```

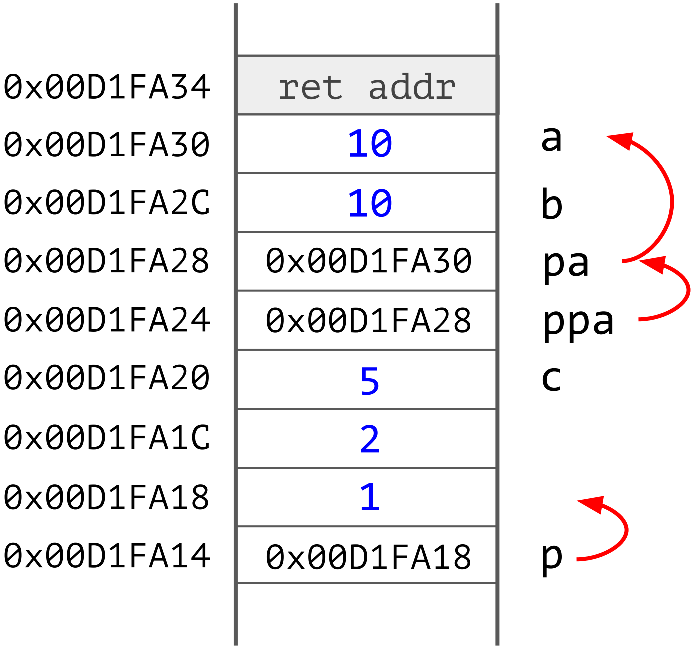


0x00D1FA34	ret addr	
0x00D1FA30	10	a
0x00D1FA2C	10	b
0x00D1FA28	0x00D1FA30	pa
0x00D1FA24	0x00D1FA28	ppa
0x00D1FA20	5	c
0x00D1FA1C	2	
0x00D1FA18	1	
0x00D1FA14	0x00D1FA18	p

let it be 32-bit architecture in this example, just to keep it simple

Pointers

```
int a = 5;
int b = 10;

int* pa = &a;
int** ppa = &pa;
...

int arr[2] = {1, 2};
int* p = arr;
p = p + 1;
```

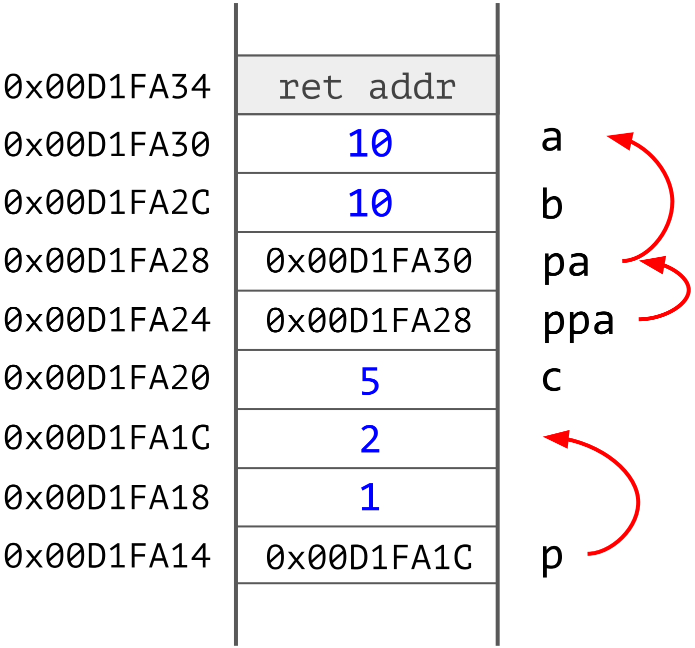


0x00D1FA34	ret addr	
0x00D1FA30	10	a
0x00D1FA2C	10	b
0x00D1FA28	0x00D1FA30	pa
0x00D1FA24	0x00D1FA28	ppa
0x00D1FA20	5	c
0x00D1FA1C	2	
0x00D1FA18	1	
0x00D1FA14	0x00D1FA18	p

let it be 32-bit architecture in this example, just to keep it simple

Pointers

```
int a = 5;
int b = 10;

int* pa = &a;
int** ppa = &pa;
...

int arr[2] = {1, 2};
int* p = arr;
p = p + 1;
```



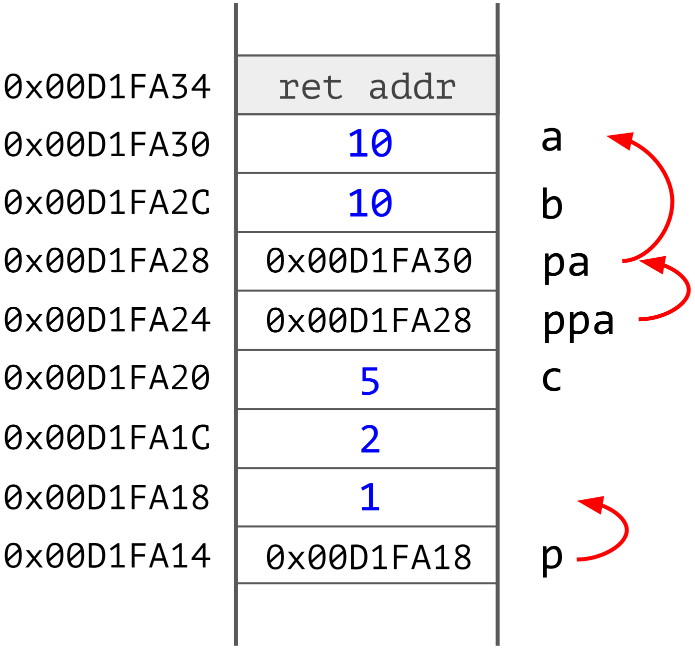
0x00D1FA34	ret addr	
0x00D1FA30	10	a
0x00D1FA2C	10	b
0x00D1FA28	0x00D1FA30	pa
0x00D1FA24	0x00D1FA28	ppa
0x00D1FA20	5	c
0x00D1FA1C	2	
0x00D1FA18	1	
0x00D1FA14	0x00D1FA1C	p

pointer arithmetics: +sizeof(int)

Pointers

let it be 32-bit architecture in this example, just to keep it simple

```
int a = 5;
int b = 10;

int* pa = &a;
int** ppa = &pa;
...

int arr[2] = {1, 2};
int* p = arr;
p[1] = 13;
```

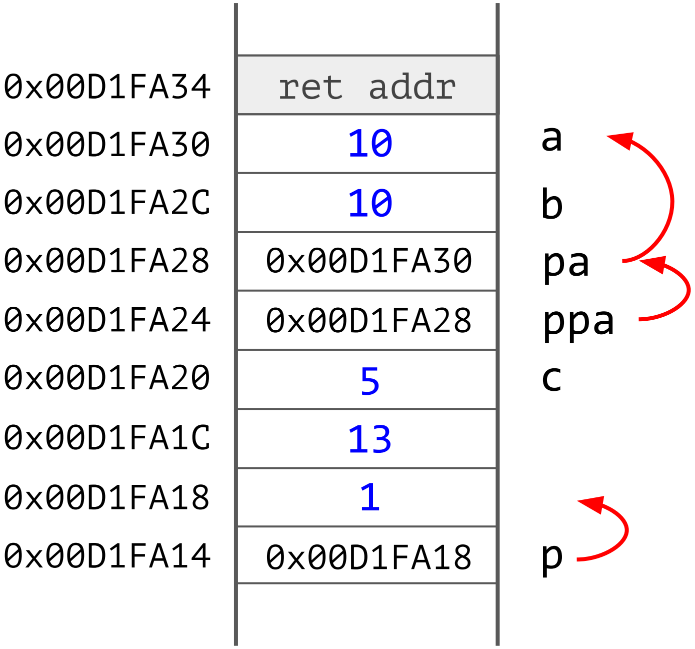


0x00D1FA34	ret addr	
0x00D1FA30	10	a
0x00D1FA2C	10	b
0x00D1FA28	0x00D1FA30	pa
0x00D1FA24	0x00D1FA28	ppa
0x00D1FA20	5	c
0x00D1FA1C	2	
0x00D1FA18	1	
0x00D1FA14	0x00D1FA18	p

let it be 32-bit architecture in this example, just to keep it simple

Pointers

```
int a = 5;
int b = 10;

int* pa = &a;
int** ppa = &pa;
...

int arr[2] = {1, 2};
int* p = arr;
*(p + 1) = 13;
```

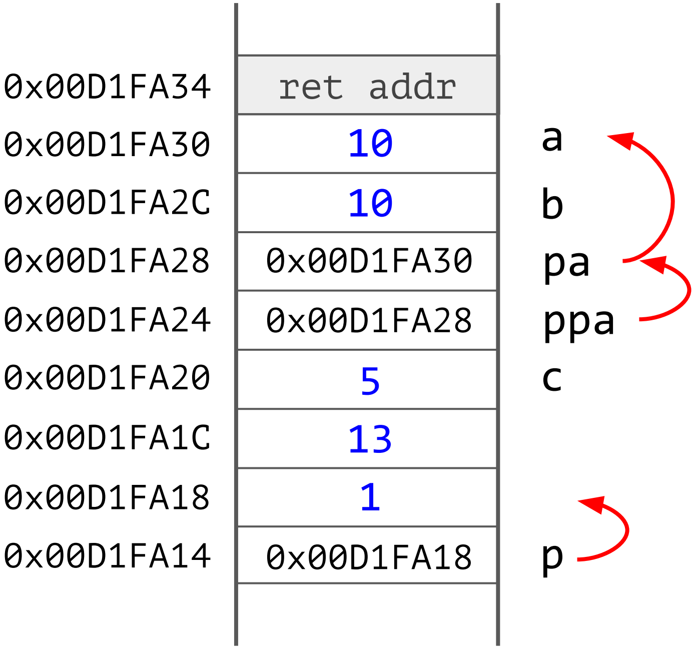


0x00D1FA34	ret addr	
0x00D1FA30	10	a
0x00D1FA2C	10	b
0x00D1FA28	0x00D1FA30	pa
0x00D1FA24	0x00D1FA28	ppa
0x00D1FA20	5	c
0x00D1FA1C	13	
0x00D1FA18	1	
0x00D1FA14	0x00D1FA18	p

Pointers (a bit crazy)

let it be 32-bit architecture in
this example, just to keep it simple

```
int a = 5;
int b = 10;
...

int arr[2] = {1, 2};
int* p = arr;
p[1] = 13;
```

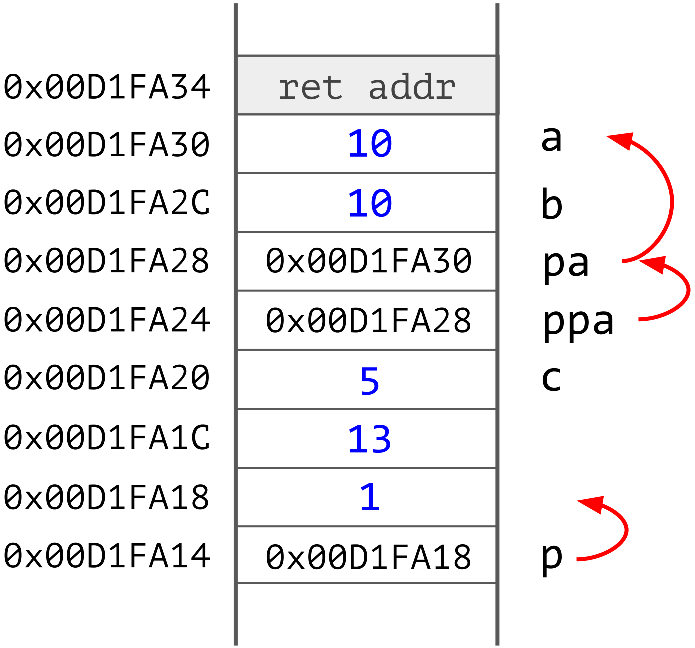


0x00D1FA34	ret addr	
0x00D1FA30	10	a
0x00D1FA2C	10	b
0x00D1FA28	0x00D1FA30	pa
0x00D1FA24	0x00D1FA28	ppa
0x00D1FA20	5	c
0x00D1FA1C	13	
0x00D1FA18	1	
0x00D1FA14	0x00D1FA18	p

Pointers (a bit crazy)

let it be 32-bit architecture in
this example, just to keep it simple

```
int a = 5;
int b = 10;
...

int arr[2] = {1, 2};
int* p = arr;
p[1] = 13;
0[p] = 19; // ???
```

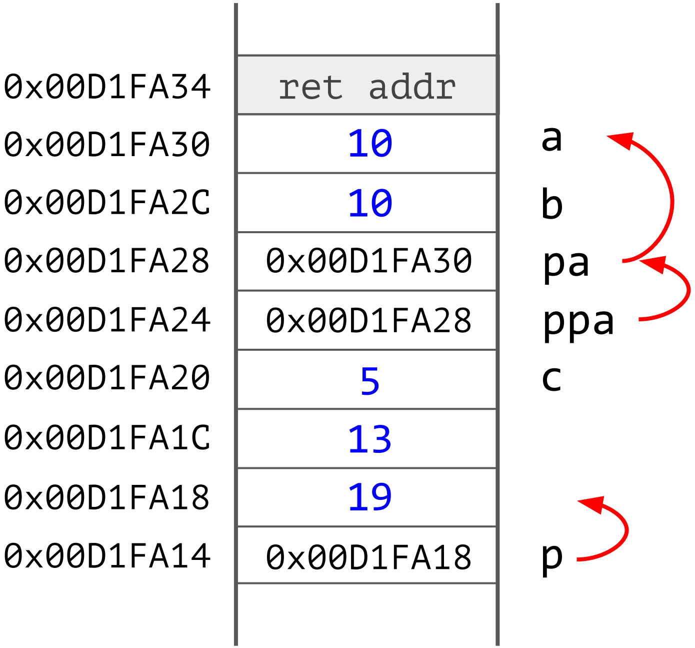


0x00D1FA34	ret addr	
0x00D1FA30	10	a
0x00D1FA2C	10	b
0x00D1FA28	0x00D1FA30	pa
0x00D1FA24	0x00D1FA28	ppa
0x00D1FA20	5	c
0x00D1FA1C	13	
0x00D1FA18	1	
0x00D1FA14	0x00D1FA18	p

let it be 32-bit architecture in
this example, just to keep it simple

Pointers (a bit crazy)

```
int a = 5;
int b = 10;
...

int arr[2] = {1, 2};
int* p = arr;
p[1] = 13;
*(0 + p) = 19;
```

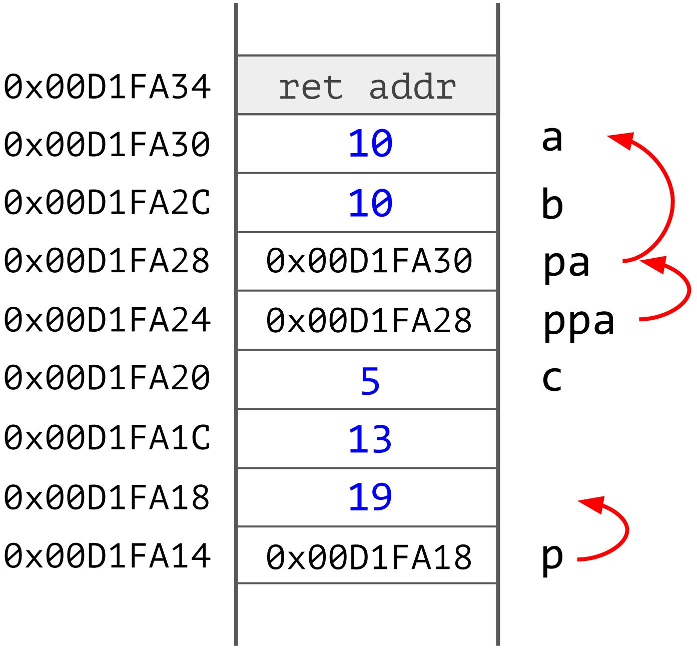


0x00D1FA34	ret addr	
0x00D1FA30	10	a
0x00D1FA2C	10	b
0x00D1FA28	0x00D1FA30	pa
0x00D1FA24	0x00D1FA28	ppa
0x00D1FA20	5	c
0x00D1FA1C	13	
0x00D1FA18	19	
0x00D1FA14	0x00D1FA18	p

Pointers (a bit crazy)

let it be 32-bit architecture in this example, just to keep it simple

```
int a = 5;
int b = 10;
...

int arr[2] = {1, 2};
int* p = arr;
p[1] = 13;
0[p] = 19;
```

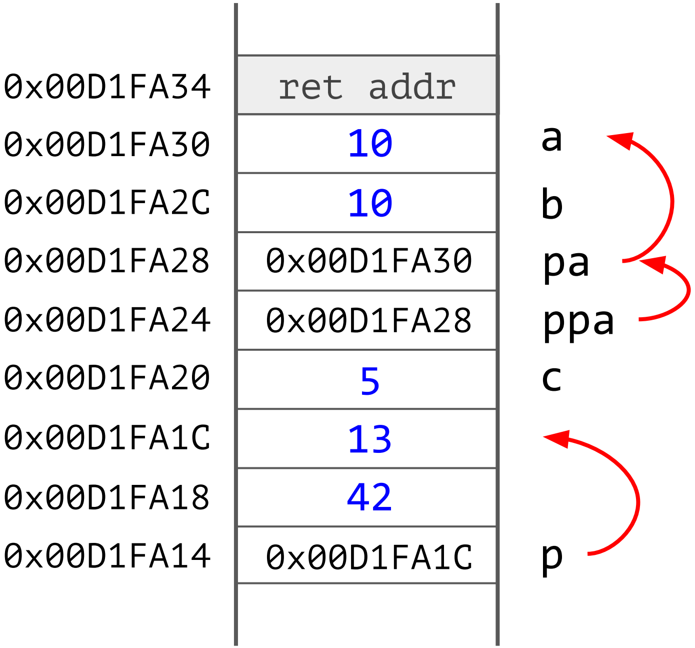


0x00D1FA34	ret addr	
0x00D1FA30	10	a
0x00D1FA2C	10	b
0x00D1FA28	0x00D1FA30	pa
0x00D1FA24	0x00D1FA28	ppa
0x00D1FA20	5	c
0x00D1FA1C	13	
0x00D1FA18	19	
0x00D1FA14	0x00D1FA18	p

Pointers (a bit crazy)

let it be 32-bit architecture in this example, just to keep it simple

```
int a = 5;
int b = 10;
...

int arr[2] = {1, 2};
int* p = arr;
p[1] = 13;
0[p] = 19;
p = arr + 1;
p[-1] = 42;
```



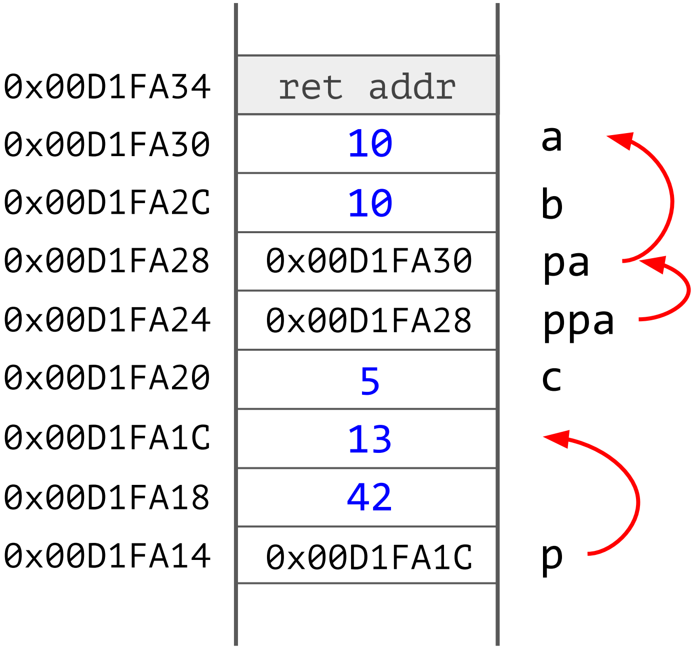
0x00D1FA34	ret addr	
0x00D1FA30	10	a
0x00D1FA2C	10	b
0x00D1FA28	0x00D1FA30	pa
0x00D1FA24	0x00D1FA28	ppa
0x00D1FA20	5	c
0x00D1FA1C	13	
0x00D1FA18	42	
0x00D1FA14	0x00D1FA1C	p

well, why not…

Pointers (a bit crazy)

let it be 32-bit architecture in this example, just to keep it simple

```
int a = 5;
int b = 10;

int* pa = &a;
int** ppa = &pa;

int c = *pa;
*pa = b;
...
```

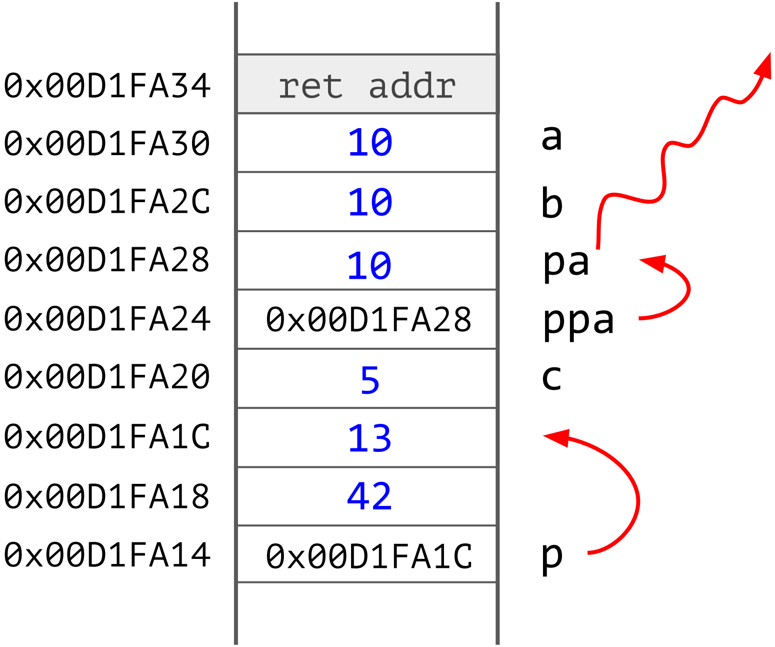


0x00D1FA34	ret addr	
0x00D1FA30	10	a
0x00D1FA2C	10	b
0x00D1FA28	0x00D1FA30	pa
0x00D1FA24	0x00D1FA28	ppa
0x00D1FA20	5	c
0x00D1FA1C	13	
0x00D1FA18	42	
0x00D1FA14	0x00D1FA1C	p

let it be 32-bit architecture in this example, just to keep it simple

Pointers (a bit crazy)

```
int a = 5;
int b = 10;

int* pa = &a;
int** ppa = &pa;

int c = *pa;
*pa = b;
...
pa = a;
```

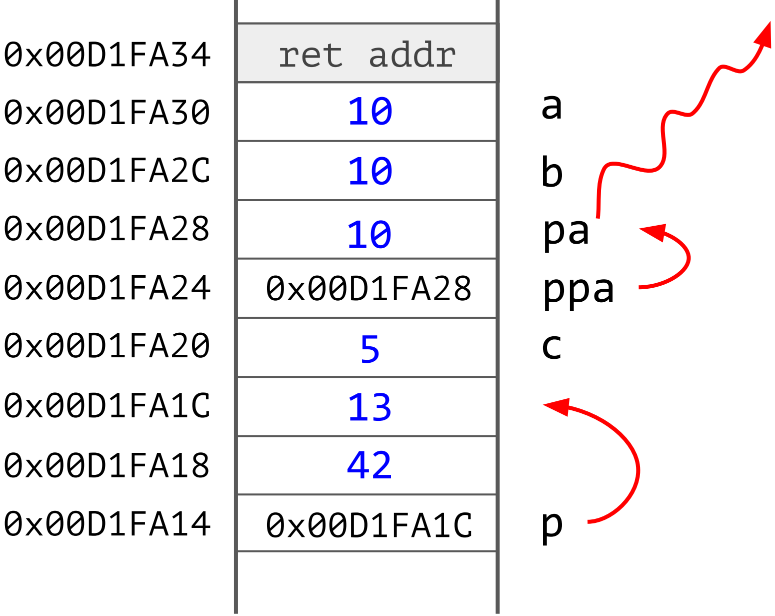


0x00D1FA34	ret addr	
0x00D1FA30	10	a
0x00D1FA2C	10	b
0x00D1FA28	10	pa
0x00D1FA24	0x00D1FA28	ppa
0x00D1FA20	5	c
0x00D1FA1C	13	
0x00D1FA18	42	
0x00D1FA14	0x00D1FA1C	p

let it be 32-bit architecture in
this example, just to keep it simple

Pointers (a bit crazy)

```
int a = 5;
int b = 10;

int* pa = &a;
int** ppa = &pa;

int c = *pa;
*pa = b;
...
pa = a;
*pa = 42;
```



0x00D1FA34	ret addr	
0x00D1FA30	10	a
0x00D1FA2C	10	b
0x00D1FA28	10	pa
0x00D1FA24	0x00D1FA28	ppa
0x00D1FA20	5	c
0x00D1FA1C	13	
0x00D1FA18	42	
0x00D1FA14	0x00D1FA1C	p

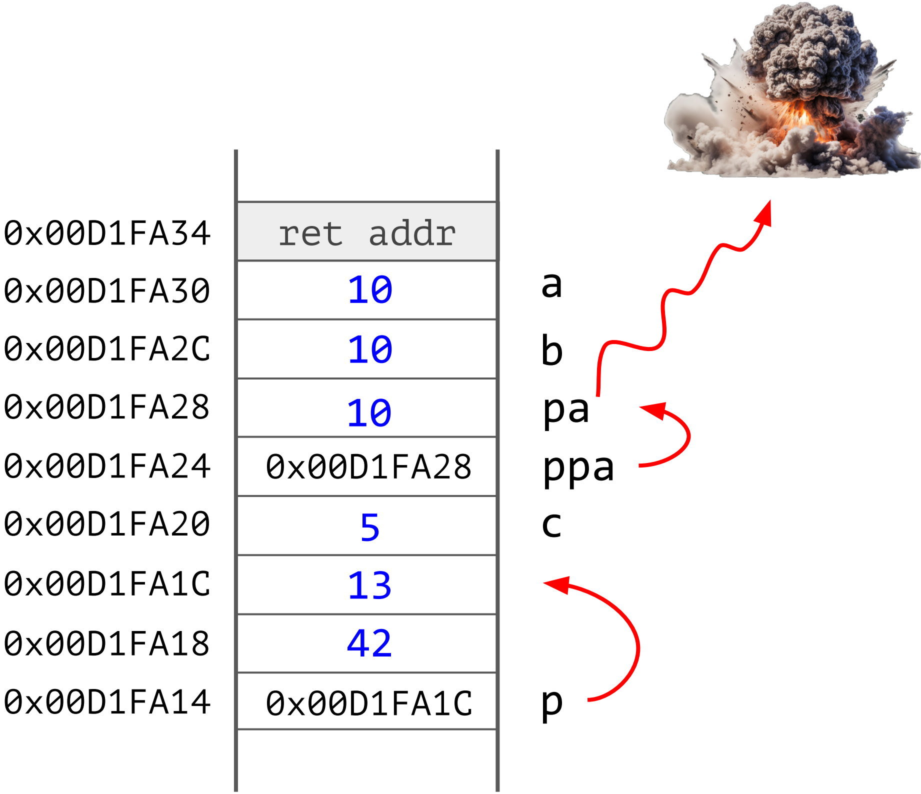
Pointers are unsafe!

Pointers

let it be 32-bit architecture in this example, just to keep it simple

Pointers : update in C++

```
int a = 5;
int b = 10;

int* pa = &a;
int** ppa = &pa;

int c = *pa;
*pa = b;
...
pa = a;
*pa = 42;
```



0x00D1FA34	ret addr	
0x00D1FA30	10	a
0x00D1FA2C	10	b
0x00D1FA28	10	pa
0x00D1FA24	0x00D1FA28	ppa
0x00D1FA20	5	c
0x00D1FA1C	13	
0x00D1FA18	42	
0x00D1FA14	0x00D1FA1C	p

Pointers : update in C++

let it be 32-bit architecture in this example, just to keep it simple

```
int a = 5;
int b = 10;

int* pa = &a;
int** ppa = &pa;

int c = *pa;
*pa = b;
...
pa = (int*) a;
*pa = 42;
```

in C++ you at least need a cast

0x00D1FA34	ret addr	
0x00D1FA30	10	a
0x00D1FA2C	10	b
0x00D1FA28	10	pa
0x00D1FA24	0x00D1FA28	ppa
0x00D1FA20	5	c
0x00D1FA1C	13	
0x00D1FA18	42	
0x00D1FA14	0x00D1FA1C	p

Pointers : update in C++

How to define a pointer that points to nowhere in C?

Pointers : update in C++

How to define a pointer that points to nowhere in C?

```
int* p1 = 0;
```

this is correct way, explicitly mentioned in specification, but a bit confusing (even for C)

Pointers : update in C++

How to define a pointer that points to nowhere in C?

```
int* p1 = 0;
```

```
int* p2 = NULL;
```

Better way. What is the type of NULL?

Pointers : update in C++

How to define a pointer that points to nowhere in C?

```
int* p1 = 0;

int* p2 = NULL;
```

Better way. What is the type of NULL?

void*! It makes it harder to assign it to some integer vars (compiler warnings)

Pointers : update in C++

How to define a pointer that points to nowhere in C++?

```
int* p1 = ???;
```

Pointers : update in C++

How to define a pointer that points to nowhere in C++?

```
int* p1 = 0;
```

Still correct! But confusing as a hell (assignments of other ints are prohibited)

Pointers : update in C++

How to define a pointer that points to nowhere in C++?

```
int* p1 = 0;
```

Still correct! But confusing as a hell (assignments of other ints are prohibited)

```
int* p2 = nullptr;
```

Pointers : update in C++

How to define a pointer that points to nowhere in C++?

```
int* p1 = 0;
```

Still correct! But confusing as a hell (assignments of other ints are prohibited)

```
int* p2 = nullptr;
```

The only value of special type nullptr_t. Implicitly converted to the pointer of needed type.

Pointers : update in C++

How to define a pointer that points to nowhere in C++?

```
int* p1 = 0;
```

Still correct! But confusing as a hell (assignments of other ints are prohibited)

```
int* p2 = nullptr;
```

The only value of special type nullptr_t. Implicitly converted to the pointer of needed type.

Heavily needed for overloading, will discuss it later!

Pointers : update in C++

Also, pointers are returned by `new` operators and consumed by `delete` operators.

Pointers : update in C++

Also, pointers are returned by new operators and consumed by delete operators.

As usual, it is unsafe and ca be used in a wrong way.

```
int* p = new int;

delete p; // ok
delete p; // double freeing => UB
```

Pointers : update in C++

Also, pointers are returned by new operators and consumed by delete operators.

As usual, it is unsafe and ca be used in a wrong way.

```
int* p = new int;

delete p; // ok
delete p; // double freeing => UB
```

Use sanitizers to detect and fix such problems!

Pointers : update in C++

How to define a pointer that points to nowhere in C?

```
int* p1 = 0;

int* p2 = NULL;   Better way. What is the type of NULL?
```

Pointers: one more typical use

```
void swap(int a, int b) {
   int c = b;
   b = a;
   a = c;
}
```

a, b - local variables in swap, copied on call

Pointers: one more typical use

```
void swap(int a, int b) {
   int c = b;
   b = a;
   a = c;
}
```

a, b - local variables in swap, copied on call

```
void main(){
   int k = 5;
   int l = 10;
   swap(k, l);
   cout << "k= " << k << " l= " << l;
}
```

stack frame of swap cleared after return, k and l are unchanged

Pointers: one more typical use

```
void swap(int* pa, int* pb) {
   int c = *pb;
   *pb = *pa;
   *pa = c;
}
```

a, b - local variables of type int* in swap, addresses copied on call

```
void main(){
   int k = 5;
   int l = 10;
   swap(&k, &l);
   cout << "k= " << k << " l= " << l;
}
```

stack frame of swap cleared after return, k and l updated as we worked with their addresses, not values

Pointers: discussion

```
Pointers are good for:
```

Pointers: discussion

Pointers are good for:

- Direct manipulations with memory

Pointers: discussion

Pointers are good for:

- Direct manipulations with memory

- Low-level stuff where pointers arithmetic works nice (compressing data for example)

Pointers: discussion

Pointers are good for:

- Direct manipulations with memory
- Low-level stuff where pointers arithmetic works nice (compressing data for example)
- Hacks!

Pointers: discussion

Pointers are good for:

- Direct manipulations with memory
- Low-level stuff where pointers arithmetic works nice (compressing data for example)
- Hacks!

Pointers are bad for:

Pointers: discussion

Pointers are good for:

- Direct manipulations with memory
- Low-level stuff where pointers arithmetic works nice (compressing data for example)
- Hacks!

Pointers are bad for:

- Everything else!

Pointers: discussion

Pointers are good for:

- Direct manipulations with memory
- Low-level stuff where pointers arithmetic works nice (compressing data for example)
- Hacks!

Pointers are bad for:

- Everything else!

Because they are:

- UNSAFE
- Too verbose

Pointers: one more typical use

```
void swap(int* pa, int* pb) {
   int c = *pb;
   *pb = *pa;
   *pa = c;
}
```

a, b - local variables of type int* in swap, addresses copied on call

---

```
void main(){
   int k = 5;
   int l = 10;
   swap(&k, &l);
   cout << "k= " << k << " l= " << l;
}
```

stack frame of swap cleared after return, k and l updated as we worked with their addresses, not values

Pointers: discussion

Pointers are good for:

- Direct manipulations with memory
- Low-level stuff where pointers arithmetic works nice (compressing data for example)
- Hacks!

Pointers are bad for:

- Everything else!

Because they are:

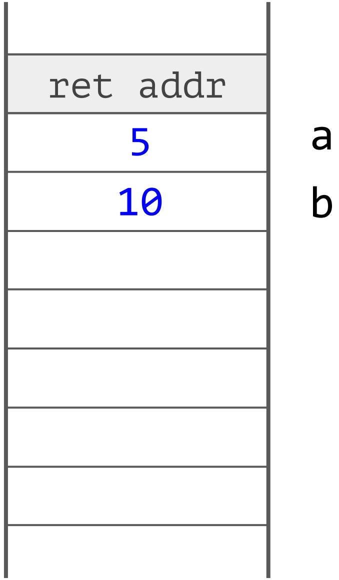
- UNSAFE
- Too verbose

Pointers look like too low-level and often misused stuff.

References

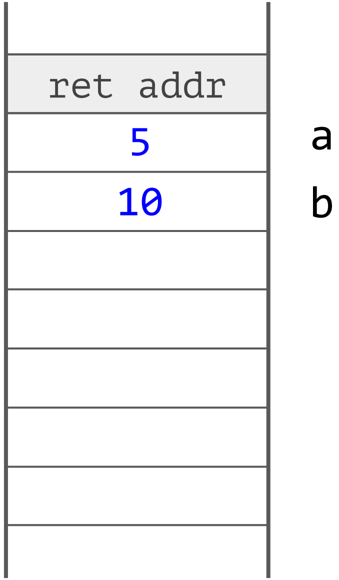
References

```
int a = 5;
int b = 10;
```



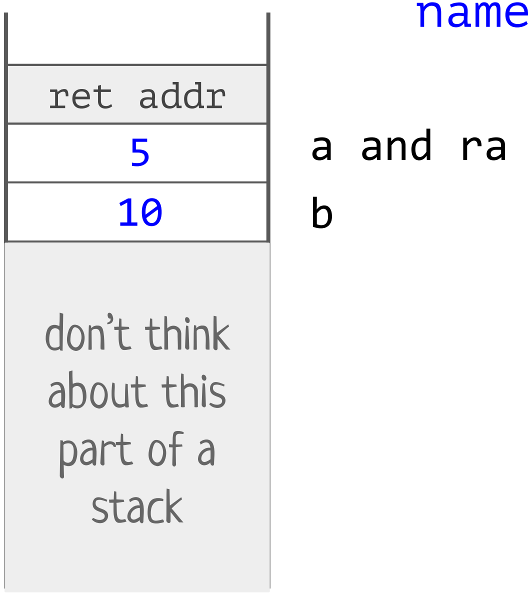
References

```
int a = 5;
int b = 10;

int& ra = a;
```



References

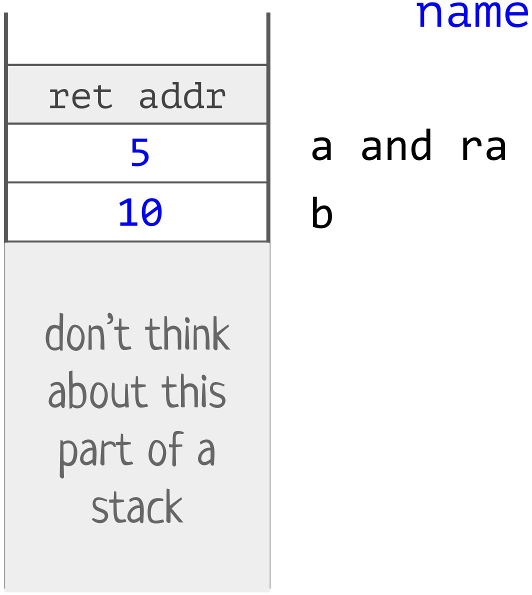
```
int a = 5;
int b = 10;

int& ra = a;
```



just another name for a!

References

```
int a = 5;
int b = 10;

int& ra = a;

ra = 42;
ra += 5;
b = ra - 1;
```



ret addr

5

10

don't think about this part of a stack

a and ra

b

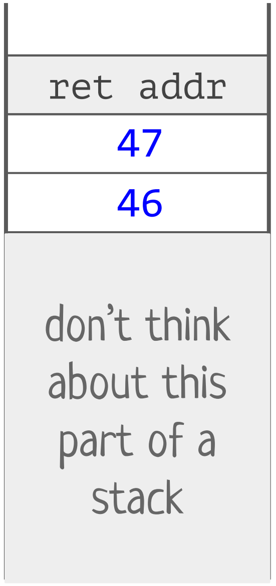
just another name for a!

References

```
int a = 5;
int b = 10;

int& ra = a;

ra = 42;
ra += 5;
b = ra - 1;
```

everything you do with ra affects a

just another name for a!



ret addr

47

46

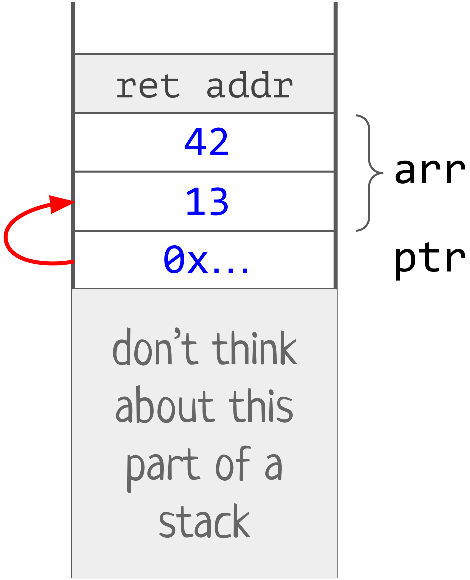
don't think about this part of a stack

a and ra

b

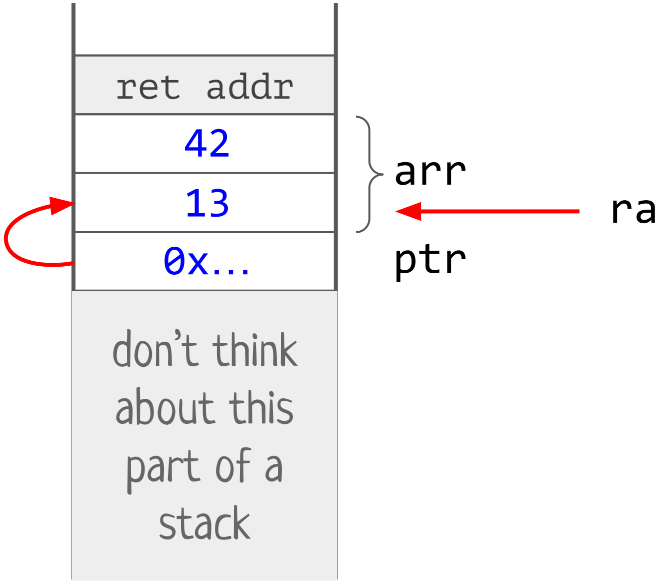
References

```
int arr[2] = {13, 42};
int* ptr = &arr[0];

int& ra = arr[0];
```



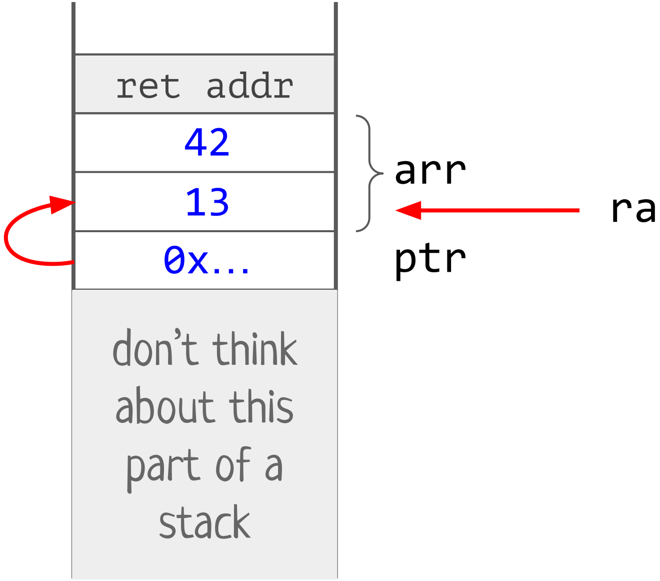
References

```
int arr[2] = {13, 42};
int* ptr = &arr[0];

int& ra = arr[0];
```



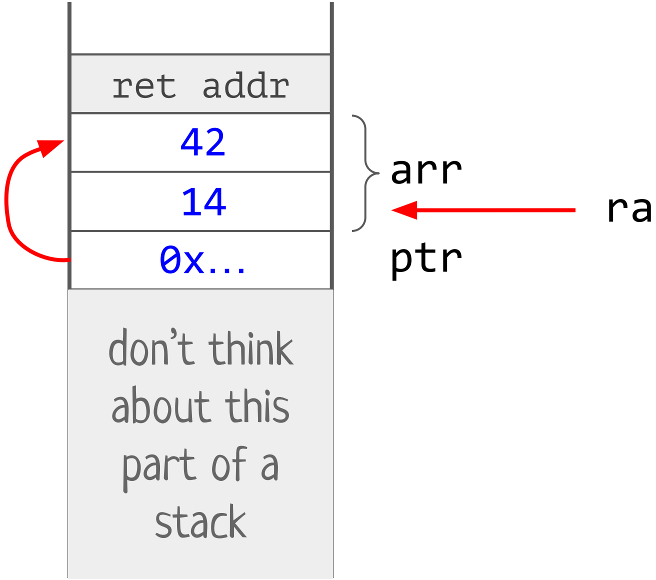
References

```
int arr[2] = {13, 42};
int* ptr = &arr[0];

int& ra = arr[0];

ptr += 1;
ra  += 1;
```



References

```
int arr[2] = {13, 42};
int* ptr = &arr[0];

int& ra = arr[0];

ptr += 1;
ra  += 1;
```

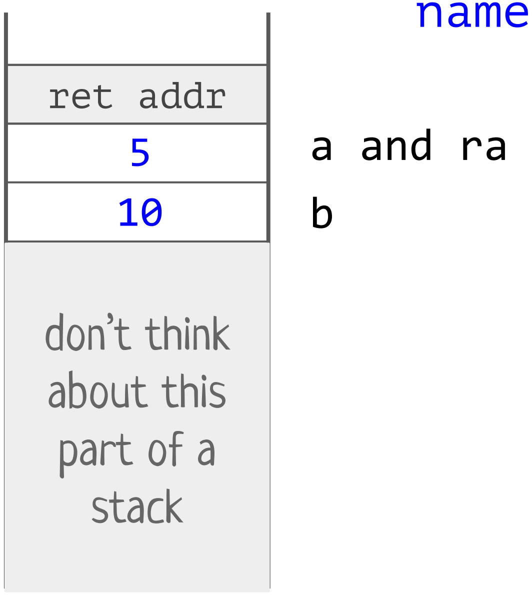


References

Doesn’t it look
somehow familiar?

```
int a = 5;
int b = 10;

int& ra = a;

ra = 42;
ra += 5;
b = ra - 1;
```

just another
name for a!



ret addr

5

10

a and ra

b

don’t think about this part of a stack

References

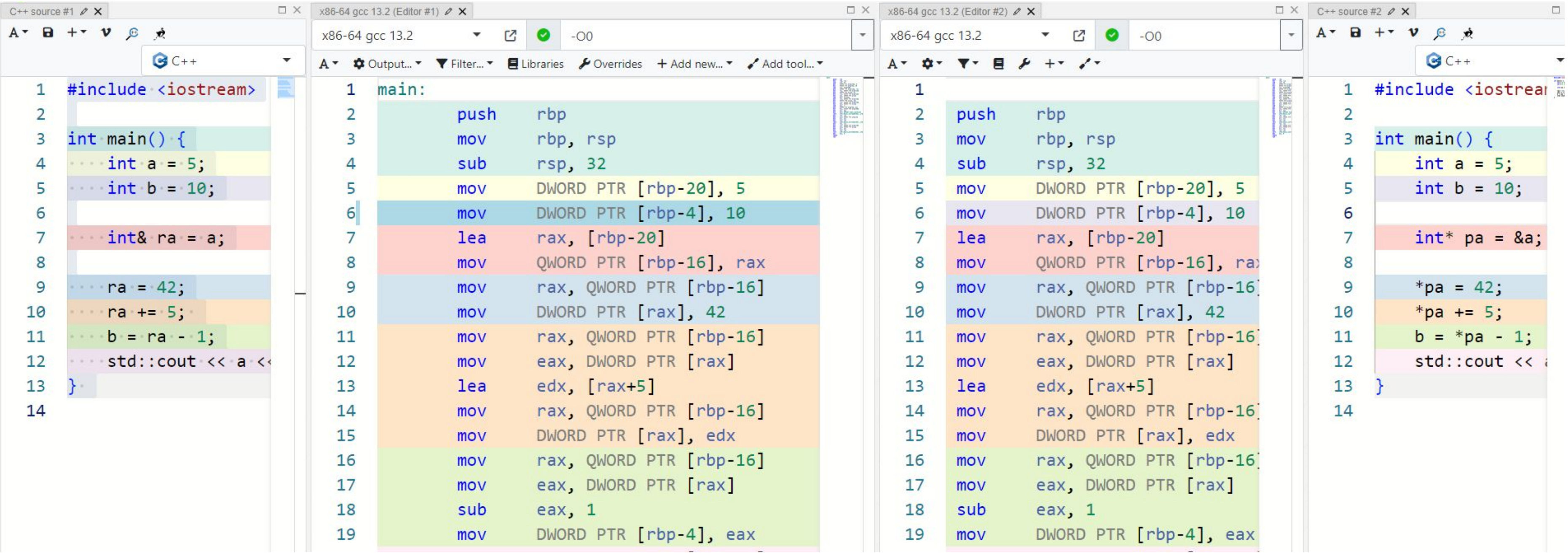
Doesn't it look somehow familiar?

```
int a = 5;
int b = 10;

int& ra = a;

ra = 42;
ra += 5;
b = ra - 1;
```

```
int a = 5;
int b = 10;

int* pa = &a;

*pa = 42;
*pa += 5;
b = *pa - 1;
```

https://godbolt.org/z/8rYbzE648

https://godbolt.org/z/8rYbzE648



x86-64 gcc 13.2 -O0

```
#include <iostream>

int main() {
    int a = 5;
    int b = 10;

    int& ra = a;

    ra = 42;
    ra += 5;
    b = ra - 1;
    std::cout << a <<
}
```

```
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     DWORD PTR [rbp-20], 5
        mov     DWORD PTR [rbp-4], 10
        lea     rax, [rbp-20]
        mov     QWORD PTR [rbp-16], rax
        mov     rax, QWORD PTR [rbp-16]
        mov     DWORD PTR [rax], 42
        mov     rax, QWORD PTR [rbp-16]
        mov     eax, DWORD PTR [rax]
        lea     edx, [rax+5]
        mov     rax, QWORD PTR [rbp-16]
        mov     DWORD PTR [rax], edx
        mov     rax, QWORD PTR [rbp-16]
        mov     eax, DWORD PTR [rax]
        sub     eax, 1
        mov     DWORD PTR [rbp-4], eax
```

x86-64 gcc 13.2 -O0

```

push    rbp
mov     rbp, rsp
sub     rsp, 32
mov     DWORD PTR [rbp-20], 5
mov     DWORD PTR [rbp-4], 10
lea     rax, [rbp-20]
mov     QWORD PTR [rbp-16], ra
mov     rax, QWORD PTR [rbp-16
mov     DWORD PTR [rax], 42
mov     rax, QWORD PTR [rbp-16
mov     eax, DWORD PTR [rax]
lea     edx, [rax+5]
mov     rax, QWORD PTR [rbp-16
mov     DWORD PTR [rax], edx
mov     rax, QWORD PTR [rbp-16
mov     eax, DWORD PTR [rax]
sub     eax, 1
mov     DWORD PTR [rbp-4], eax
```

```
#include <iostrea

int main() {
    int a = 5;
    int b = 10;

    int* pa = &a;

    *pa = 42;
    *pa += 5;
    b = *pa - 1;
    std::cout <<
}
```

References

References are implemented as hidden pointers. But they are much safer.

```
int a = 5;                      int a = 5;
int b = 10;                     int b = 10;

int& ra = a;          ↔         int* pa = &a;

ra = 42;              ↔         *pa = 42;
ra += 5;              ↔         *pa += 5;
b = ra - 1;           ↔         b = *pa - 1;
```

References: safety

```
int a = 5;

int& ra; // compilation
         // error
```

1. Any reference should be initialized.

References: safety

```
int a = 5;

int& ra = a; // ok
```

1. Any reference should be initialized.

References: safety

```
int a = 5;

int& ra = a; // ok
```

1. Any reference should be initialized.

2. Once initialized, it can't be "redirected" to another object.

References: safety

```
int a = 5;
int b = 10;

int& ra = a; // ok

ra = b; // you are just
        // setting "a"
        // into value
        // of "b"
```

1. Any reference should be initialized.
2. Once initialized, it can’t be "redirected" to another object.

References: safety

```
int a = 5;

int& ra = a; // ok
```

1. Any reference should be initialized.

2. Once initialized, it can’t be "redirected" to another object.

References: safety

```
int a = 5;

int& ra = a; // ok
```

1. Any reference should be initialized.

2. Once initialized, it can’t be "redirected" to another object.

3. References are not objects. They do not have addresses; no references over reference; no arrays of references.

References: safety

```
int a = 5;
int& ra = a; // ok

int* pa = &a;
int*& rpa = pa;
```

1. Any reference should be initialized.

2. Once initialized, it can’t be "redirected" to another object.

3. References are not objects. They do not have addresses; no references over reference; no arrays of references.

References: safety

```
int a = 5;
int& ra = a; // ok

int* pa = &a;
int*& rpa = pa; // ok!
// it was just a
// reference to pointer
```

1. Any reference should be initialized.
2. Once initialized, it can’t be "redirected" to another object.
3. References are not objects. They do not have addresses; no references over reference; no arrays of references.

References: safety

```
int a = 5;
int& ra = a; // ok

int* pa = &a;
int*& rpa = pa; // ok!

int&* pra = &pa;
```

1. Any reference should be initialized.
2. Once initialized, it can't be "redirected" to another object.
3. References are not objects. They do not have addresses; no references over reference; no arrays of references.

References: safety

```
int a = 5;
int& ra = a; // ok

int* pa = &a;
int*& rpa = pa; // ok!

int&* pra = &pa;

// error: cannot declare
pointer to 'int&'
```

1. Any reference should be initialized.
2. Once initialized, it can’t be "redirected" to another object.
3. References are not objects. They do not have addresses; no references over reference; no arrays of references.

References: safety

```
int a = 5;
int& ra = a; // ok

int* pa = &a;
int*& rpa = pa; // ok!
```

1. Any reference should be initialized.

2. Once initialized, it can’t be "redirected" to another object.

3. References are not objects. They do not have addresses; no references over reference; no arrays of references.

4. No pointer (reference) arithmetic.

References: safety

```
int a = 5;
int& ra = a; // ok

int* pa = &a;
int*& rpa = pa; // ok!

ra += 1;
// just increments a
```

1. Any reference should be initialized.
2. Once initialized, it can’t be "redirected" to another object.
3. References are not objects. They do not have addresses; no references over reference; no arrays of references.
4. No pointer (reference) arithmetic.

References: safety

```
int a = 5;
int& ra = a; // ok

int* pa = &a;
int*& rpa = pa; // ok!

ra += 1;
// just increments a
```

1. Any reference should be initialized.

2. Once initialized, it can’t be "redirected" to another object.

3. References are not objects. They do not have addresses; …

4. No pointer (reference) arithmetic.

5. No delete from ref!

Pointers: one more typical use

```
void swap(int* pa, int* pb) {
   int c = *pb;
   *pb = *pa;
   *pa = c;
}
```

a, b - local variables of type int* in swap, addresses copied on call

---

```
void main(){
   int k = 5;
   int l = 10;
   swap(&k, &l);
   cout << "k= " << k << " l= " << l;
}
```

stack frame of swap cleared after return, k and l updated as we worked with their addresses, not values

Pointers: one more typical use

```
void swap(int* pa, int* pb) {
   int c = *pb;
   delete pb;
   *pb = *pa;
   *pa = c;
}
```

if function works with pointers, no-one will stop developer from doing such crime!!!

```
void main(){
   int k = 5;
   int l = 10;
   swap(&k, &l);
   cout << "k= " << k << " l= " << l;
}
```

stack frame of swap cleared after return, k and l updated as we worked with their addresses, not values

Pointers: one more typical use

```
void swap(int* pa, int* pb) {
   int c = *pb;
   *pb = *pa;
   *pa = c;
}



void main(){
   int k = 5;
   int l = 10;
   swap(&k, &l);
   cout << "k= " << k << " l= " << l;
}
```

References: one more typical use

```
void swap(int& pa, int& pb) {
   int c = pb;
   pb = pa;
   pa = c;
}



void main(){
   int k = 5;
   int l = 10;
   swap(k, l);
   cout << "k= " << k
        << " l= " << l;
}
```

References: one more typical use

```
void swap(int& pa, int& pb) {
   int c = pb;
   pb = pa;
   pa = c;
}



void main(){
   int k = 5;
   int l = 10;
   swap(k, l);
   cout << "k= " << k
        << " l= " << l;
}
```

Pros:

1. Less boilerplate

2. Safer: fewer ways to shoot yourself to the foot

References: one more typical use

```
void swap(int& pa, int& pb) {
   int c = pb;
   delete pa; // compilation
              // error
   pb = pa;
   pa = c;
}

void main(){
   int k = 5;
   int l = 10;
   swap(k, l);
   cout << "k= " << k
        << " l= " << l;
}
```

Pros:

1. Less boilerplate
2. Safer: fewer ways to shoot yourself to the foot

References: one more typical use

```
void swap(int& pa, int& pb) {
   int c = pb;
   pb = pa;
   pa = c;
}


void main(){
   int k = 5;
   int l = 10;
   swap(k, l);
   cout << "k= " << k
        << " l= " << l;
}
```

Pros:

1. Less boilerplate
2. Safer: fewer ways to shoot yourself to the foot
3. Zero-overhead abstraction

References: one more typical use

```
void swap(int& pa, int& pb) {
   int c = pb;
   pb = pa;
   pa = c;
}


void main(){
   int k = 5;
   int l = 10;
   swap(k, l);
   cout << "k= " << k
        << " l= " << l;
}
```

Pros:

1. Less boilerplate
2. Safer: fewer ways to shoot yourself to the foot
3. Zero-overhead abstraction

Cons?

References: one more typical use

```
void swap(int& pa, int& pb) {
   int c = pb;
   pb = pa;
   pa = c;
}


void main(){
   int k = 5;
   int l = 10;
   swap(k, l);
   cout << "k= " << k
        << " l= " << l;
}
```

Pros:

1. Less boilerplate
2. Safer: ...
3. Zero-overhead abstraction

Cons:

1. Confusing code

References: one more typical use

```
void swap(int& pa, int& pb) {
   int c = pb;
   pb = pa;
   pa = c;
}


void main(){
   int k = 5;
   int l = 10;
   swap(k, l);
   cout << "k= " << k
        << " l= " << l;
}
```

Are these arguments references or values?

Pros:

1. Less boilerplate
2. Safer: ...
3. Zero-overhead abstraction

Cons:

1. Confusing code

References: one more typical use

```
void swap(int& pa, int& pb) {
   int c = pb;
   pb = pa;
   pa = c;
}
```

Are these arguments references or values?

```
void main(){
   int k = 5;
   int l = 10;
   swap(k, l);
   cout << "k= " << k
        << " l= " << l;
}
```

Pros:

1. Less boilerplate
2. Safer: ...
3. Zero-overhead abstraction

Cons:

1. Confusing code
2. Less powerful than pointers

References: discussion

```
int a = 5;
int& ra = a; // ok
```

So, references are implemented as pointers, but with some limited usages and restricted access to the implementation.

How do we call such thing?

References: discussion

```
int a = 5;
int& ra = a; // ok
```

So, references are implemented as pointers, but with some limited usages and restricted access to the implementation.

How do we call such thing?

Encapsulation!

References: discussion

```
int a = 5;
int& ra = a; // ok
```

So, references are implemented as pointers, but with some limited usages and restricted access to the implementation.

How do we call such thing?

Encapsulation! References are just encapsulated pointers for not low-level usages.

References: lvalue or rvalue

```
int a = 5;
int& ra = a; // ok
```

References: lvalue or rvalue

```
int a = 5;
int& ra = a; // ok

int& ref = 13;
```

What would you expect from such code? 🤔

const

const

Old good const
from C language

```
int a = 10;
const int b = 20;
a = 30;                 // ok
b = 50;                 // compilation error
const int c;            // compilation error
```

const

Old good const
from C language

```
int a = 10;
int b = 10;
const int* pa = &a;
// pointer to the constant int

(*pa) = 3; // compilation error
pa = &b;   // ok!
```

const

Old good const
from C language

```
int a = 10;
int b = 10;
int* const pa = &a;
// constant pointer to int

(*pa) = 3; // ok!
pa = &b;   // compilation error
```

const

```
int * const pa = &a;
```

const

$$\underbrace{\texttt{int}}_{\text{the pointed}}\ \texttt{*}\ \overbrace{\texttt{const pa}}^{\text{the pointer}}\ \texttt{= \&a;}$$

const

```
const int * pa = &a;
```

const

$$\underbrace{\texttt{const int}}_{\text{the pointed}}\ \texttt{*}\ \overbrace{\texttt{pa}}^{\text{the pointer}}\ \texttt{= \&a;}$$

const references

```
int a = 10;
int b = 10;
const int& ra = a;
// reference to the constant int
```

const references

```
int a = 10;
int b = 10;
const int& ra = a;
// reference to the constant int

ra = 3; // compilation error
```

const references

```
int a = 10;
int b = 10;
const int& ra = a;
// reference to the constant int

ra = 3; // compilation error
```

References themselves are always constant (you can’t redirect them)

const references

```
int a = 10;
int b = 10;
const int& ra = a;
// reference to the constant int

ra = 3; // compilation error
```

References themselves are always constant (you can’t redirect them)

That’s why we’ll pronounce "constant reference" but mean "reference to constant".

const references

```cpp
int a = 10;
int b = 10;
const int& ra = a;
// reference to the constant int

ra = 3; // compilation error
```

const is not only a restriction, but also a permission!

const references

```
const int a = 10;
int& ra = a; // compilation error
```

const references

```
const int a = 10;
int& ra = a; // compilation error

const int& rca = a; // ok
```

const member functions

```
class Vector {

    int* data_;
    size_t size_;
    size_t capacity_;

public:
    void push_back(int value) { ... }

    size_t size() const { return size_; }
};
```

what does this const mean? 🤔

const member functions

```
class Vector {

    int* data_;
    size_t size_;
    size_t capacity_;

public:
     void push_back(int value) { ... }

    size_t size() const { return size_; }
};
```

Idea is obvious: you shouldn’t modify object through such methods

const member functions

```
class Vector {

    int* data_;
    size_t size_;
    size_t capacity_;

public:
    void push_back(int value) { ... }

    size_t size() const { return size_; }
};
```

Idea is obvious: you shouldn’t modify object through such methods

```
size_t size() const {
   return size_; // ok
}
```

const member functions

```
class Vector {

    int* data_;
    size_t size_;
    size_t capacity_;

public:
    void push_back(int value) { ... }

    size_t size() const { return size_; }
};
```

Idea is obvious: you shouldn’t modify object through such methods

```
size_t size() const {
    size_++; // comp error
    return size_; // ok
}
```

const member functions

```
class Vector {

    int* data_;
    size_t size_;
    size_t capacity_;

public:
    void push_back(int value) { ... }

    size_t size() const { return size_; }
};
```

Idea is obvious: you shouldn’t modify object through such methods

```
size_t size() const {
    size_++; // comp error
    return size_; // ok
}
```

How else can we modify the object?

const member functions

```
class Vector {

    int* data_;
    size_t size_;
    size_t capacity_;

public:
    void push_back(int value) { ... }

    size_t size() const { return size_; }
};
```

Idea is obvious: you shouldn’t modify object through such methods

```
size_t size() const {
   size_++; // comp error
   push_back(13); // comp err
   return size_; // ok
}
```

Call non-const method!

const member functions

```
class Vector {

    int* data_;
    size_t size_;
    size_t capacity_;

public:
    void push_back(int value) { ... }

    size_t size() const { return size_; }
};
```

Idea is obvious: you shouldn’t modify object through such methods

```
size_t size() const {
   size_++; // comp error
   push_back(13); // comp err
   size_t& r = &size_; // err
   return size_; // ok
}
```

Getting non-const references to members!

const member functions

```
class Vector {

    int* data_;
    size_t size_;
    size_t capacity_;

public:
    void push_back(int value) { ... }

    size_t size() const { return size_; }
};
```

Idea is obvious: you shouldn’t modify object through such methods

```
size_t size() const {
    size_++; // comp error
    push_back(13); // comp err
    const size_t& r = &size_;
    return size_; // ok
}
```

const member functions

```
class Vector {

    int* data_;
    size_t size_;
    size_t capacity_;

public:
    void push_back(int value) { ... }

    size_t size() const { return size_; }
};
```

Idea is obvious: you shouldn’t modify object through such methods

```
size_t size() const {
    size_++; // comp error
    push_back(13); // comp err
    const size_t& r = &size_;
    return size_; // ok
}
```

How to implement?

const member functions

```
class Vector {

    int* data_;
    size_t size_;
    size_t capacity_;

public:
    void push_back(int value) { ... }

    size_t size() const { return size_; }
};
```

Idea is obvious: you shouldn’t modify object through such methods

```
size_t size() const {
    size_++; // comp error
    push_back(13); // comp err
    const size_t& r = &size_;
    return size_; // ok
}
```

this pointer is constant pointer to constant in such methods.

References: lvalue or rvalue

```
int a = 5;
int& ra = a; // ok

int& ref = 13;
```

What would you expect from such code? 🤔

References: lvalue or rvalue

```
int a = 5;
int& ra = a; // ok

int& ref = 13; // compilation error
```

References: lvalue or rvalue

```
int a = 5;
int& ra = a; // ok

int& ref = 13; // compilation error

const int& ref = 13; // ok
```

References: lvalue or rvalue

```
int a = 5;
int& ra = a; // ok

int& ref = 13; // compilation error

const int& ref = 13; // ok
```

Why? 🤔

Temporary objects (first look)

```
void foo(int x) {
    x += 3;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b);
```

Temporary objects (first look)

```
void foo(int x) {
    x += 3;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b);
```

What assembly code you think will be generated here?

(let it be -O0 flag)

Temporary objects (first look)

```
void foo(int x) {
    x += 3;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b);
```

What assembly code you think will be generated here?

(let it be -O0 flag)

```
push    rbp
mov     rbp, rsp
sub     rsp, 32
mov     DWORD PTR [rbp-4], 13
mov     DWORD PTR [rbp-8], 42
mov     edx, DWORD PTR [rbp-4]
mov     eax, DWORD PTR [rbp-8]
add     eax, edx
mov     edi, eax
call    foo(int)
```

Temporary objects (first look)

```
void foo(int x) {
    x += 3;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b);
```

What assembly code you think will be generated here?

(let it be -O0 flag)

```
push    rbp
mov     rbp, rsp
sub     rsp, 32
mov     DWORD PTR [rbp-4], 13  ← a
mov     DWORD PTR [rbp-8], 42  ← b
mov     edx, DWORD PTR [rbp-4]
mov     eax, DWORD PTR [rbp-8]
add     eax, edx       ← a + b in eax
mov     edi, eax
call    foo(int)
```

Temporary objects (first look)

```
void foo(int x) {
    x += 3;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b);
```

What assembly code you think will be generated here?

(let it be -O0 flag)

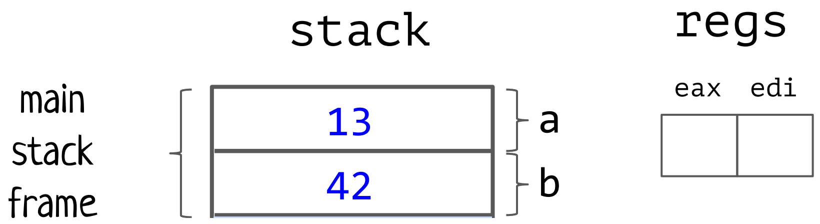
```
push    rbp
mov     rbp, rsp
sub     rsp, 32
mov     DWORD PTR [rbp-4], 13    ← a
mov     DWORD PTR [rbp-8], 42    ← b
mov     edx, DWORD PTR [rbp-4]
mov     eax, DWORD PTR [rbp-8]
add     eax, edx      ← a + b in eax
mov     edi, eax      ← copy it to 1st
call    foo(int)        argument of foo
```

Temporary objects (first look)

```
void foo(int x) {
    x += 3;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b);
```

What assembly code you think will be generated here?

(let it be -O0 flag)



stack

main stack frame

13 a

42 b

regs

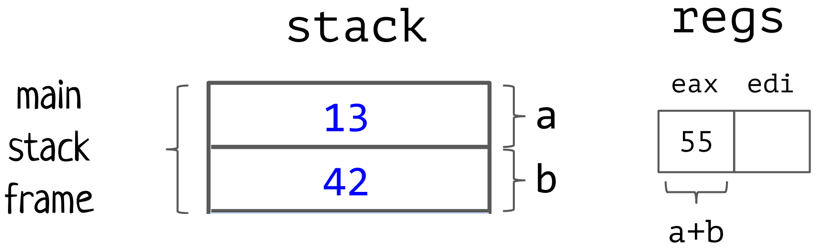
eax edi

Temporary objects (first look)

```
void foo(int x) {
    x += 3;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b);
```

What assembly code you think will be generated here?

(let it be -O0 flag)

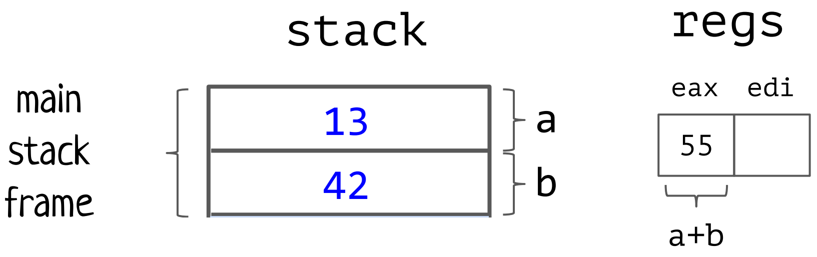


Temporary objects (first look)

```
void foo(int x) {
    x += 3;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b);
```

What assembly code you think will be generated here?

(let it be -O0 flag)



stack

regs

main stack frame

13

42

a

b

eax

edi

55

a+b

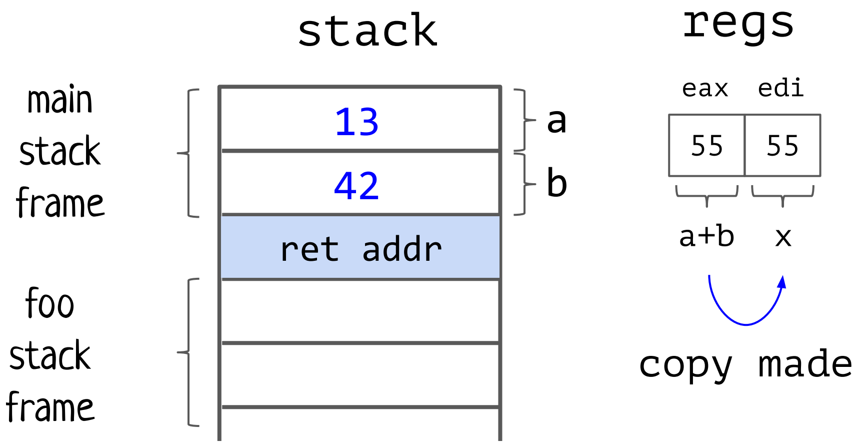
Temporary, unnamed object was created by the compiler!

Temporary objects (first look)

```
void foo(int x) {
    x += 3;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b);
```

What assembly code you think will be generated here?

(let it be -O0 flag)



stack

regs

main stack frame

13

a

42

b

ret addr

foo stack frame

eax

edi

55

55

a+b

x

copy made

Temporary objects (first look)

```
void foo(int x) {
    x += 3;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b);
```

Now imagine you have no registers on your arch!

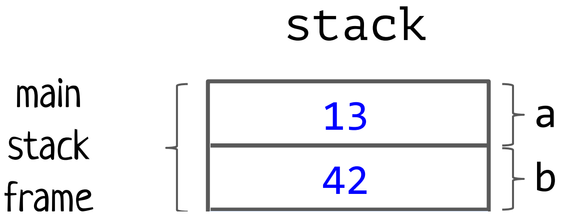
Only stack.

Temporary objects (first look)

```
void foo(int x) {
    x += 3;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b);
```

Now imagine you have no registers on your arch!

Only stack.

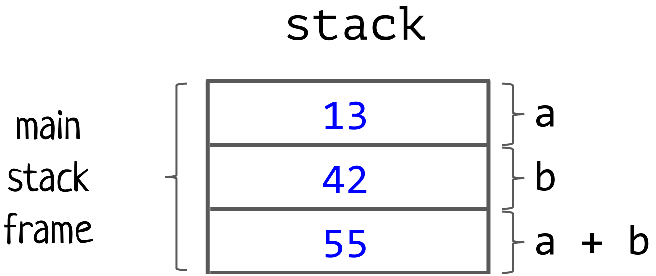


Temporary objects (first look)

```
void foo(int x) {
    x += 3;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b);
```

Now imagine you have no registers on your arch!

Only stack.

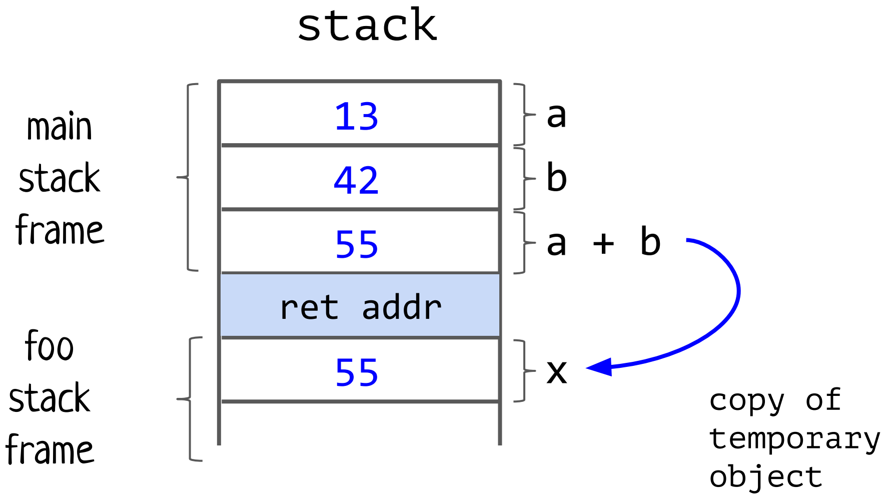


Temporary objects (first look)

```
void foo(int x) {
    x += 3;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b);
```

Now imagine you have no registers on your arch!

Only stack.



copy of temporary object

Temporary objects (first look)

Compiler can create temporary objects during its work.

Temporary objects (first look)

Compiler can create temporary objects during its work.

Those objects can be placed on the stack or in registers (or even somewhere else! see exception objs)

Temporary objects (first look)

Compiler can create temporary objects during its work.

Those objects can be placed on the stack or in registers (or even somewhere else! see exception objs)

Usually, you shouldn’t be able to get an address of such objects:

```
const int a = 13;
const int b = 42;
int* p = &(a + b); // comp error
```

Temporary objects (first look)

Compiler can create temporary objects during its work.

Those objects can be placed on the stack or in registers (or even somewhere else! see exception objs)

Usually, you shouldn't be able to get an address of such objects:

```
const int a = 13;
const int b = 42;
int* p = &(a + b); // comp error… why?
```

Temporary objects (first look)

Compiler can create temporary objects during its work.

Those objects can be placed on the stack or in registers (or even somewhere else! see exception objs)

Usually, you shouldn’t be able to get an address of such objects:

```
const int a = 13;
const int b = 42;
int* p = &(a + b); // comp error… why? because
there
                      could be no such thing! (regs)
```

Temporary objects (first look)

Compiler can create temporary objects during its work.

Those objects can be placed on the stack or in registers (or even somewhere else! see exception objs)

What is the lifetime of temporary objects?

Temporary objects (first look)

Compiler can create temporary objects during its work.

Those objects can be placed on the stack or in registers (or even somewhere else! see exception objs)

What is the lifetime of temporary objects?

```
const int a = 13;
const int b = 42;
...
foo(a + b);
```

Temporary objects (first look)

Compiler can create temporary objects during its work.

Those objects can be placed on the stack or in registers (or even somewhere else! see exception objs)

What is the lifetime of temporary objects?

```
const int a = 13;
const int b = 42;
...
foo(a + b);
```

← temporary objects alive till the end of full expression (;)

Object lifetime (first approximation, C-like)

When object dies? Depends on its storage duration:

- static => when program terminates (return from main or call exit)
- automatic => at the end of the scope
- dynamic => when delete is called

Object lifetime (second approximation)

When object dies?

If object is temporary => end of the full statement;

Otherwise, depends on its storage duration:

```
○  static    => when program terminates
                (return from main or call exit)

○  automatic => at the end of the scope

○  dynamic   => when delete is called
```

Temporary objects and references

```
void foo(int x) {
    x += 3;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b);
```

Temporary objects and references

```
void foo(int& x) {
    x += 3;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b);
```

Temporary objects and references

```cpp
void foo(int& x) {
    x += 3;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b); // compilation error
```

Temporary objects and references

```
void foo(int& x) {
    x += 3;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b); // compilation error
```

Usual references can’t be bound to temporary objects.

Temporary objects and references

```
void foo(int& x) {
    x += 3;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b); // compilation
            // error
```

Usual references can’t b
bound to temporary
objects.

The reason is that it
could be confusing: such
implicit code can lead t
changing temporary
objects what is usually
not needed.

Temporary objects and references

```
void foo(int& x) {
    x += 3;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b); // compilation
            // error
```

Usual references can’t b
bound to temporary
objects.

Such references can be
bound only to objects
with address in memory.

Temporary objects and references

```
void foo(int& x) {
    x += 3;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b); // compilation
            // error
```

Usual references can’t b
bound to temporary
objects.

Such references can be
bound only to objects
with address in memory.

That’s why they are also
called lvalue references
(initialized with lvalue
l = locator)

Temporary objects and references

```
void foo(int& x) {
   x += 3;
   ...
}

const int a = 13;
const int b = 42;
...
foo(a + b); // compilation
            // error
```

Usual references can’t b
bound to temporary
objects.

Such references can be
bound only to objects
with address in memory.

That’s why they are also
called lvalue references

Will discuss lvalue,
rvalue, gvalue, …, later

Temporary objects and references

```
void foo(int& x) {
    x += 3;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b); // compilation
            // error
```

Temporary objects and references

```
void foo(const int& x) {
   cout << x;
   ...
}

const int a = 13;
const int b = 42;
...
foo(a + b); // ok
```

Temporary objects and references

```
void foo(const int& x) {
    cout << x;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b); // ok
```

This is fine.

Temporary objects and references

```
void foo(const int& x) {
    cout << x;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b); // ok
```

Constant lvalue references can be bound to temporary objects.

Temporary objects and references

```
void foo(const int& x) {
    cout << x;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b); // ok
```

Constant lvalue references can be bound to temporary objects.

(you will not change temporary objects with them, right?)

Temporary objects and references

```
void foo(const int& x) {
    cout << x;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b); // ok
```

```
Questions:
```

Temporary objects and references

```
void foo(const int& x) {
    cout << x;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b); // ok
```

Questions:

Q: What if I try to take an address of the object reference is bound to?

Temporary objects and references

```
void foo(const int& x) {
    cout << &x;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b); // ok
```

Questions:

Q: What if I try to take an address of the object reference is bound to?

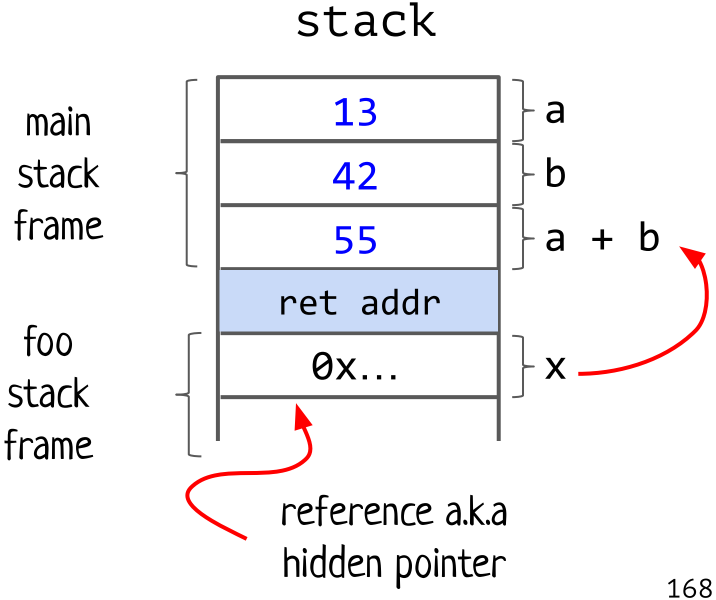
Temporary objects and references

```
void foo(const int& x) {
    cout << &x;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b); // ok
```

```
Questions:

Q: What if I try to take
an address of the object
reference is bound to?

A: You will get it.
```

Temporary objects and references

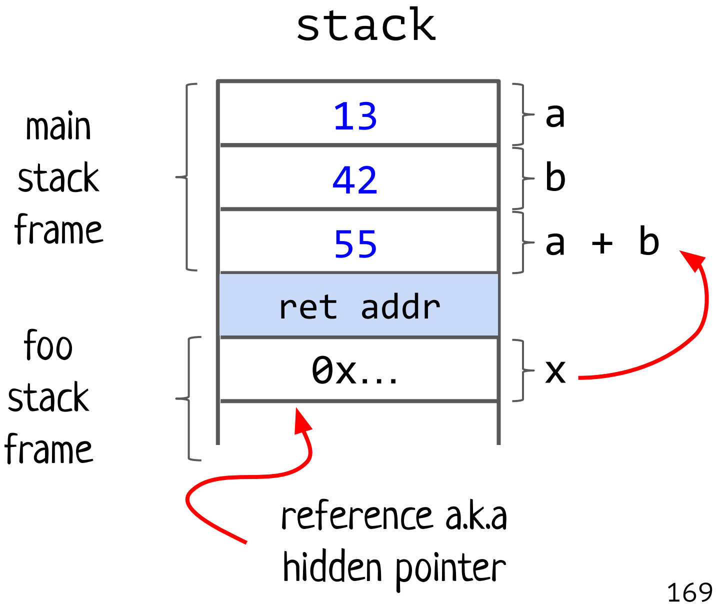
```
void foo(const int& x) {
    cout << &x;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b); // ok
```



Temporary objects and references

```
void foo(const int& x) {
    cout << &x;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b); // ok
```

Now compiler has no other choice but generate code like this!



Temporary objects and references

```
void foo(const int& x) {
    cout << &x;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b); // ok
```

Questions:

Q: Why the hell should we pass integers by const reference?

Temporary objects and references

```
void foo(const int& x) {
    cout << &x;
    ...
}

const int a = 13;
const int b = 42;
...
foo(a + b); // ok
```

Questions:

Q: Why the hell should we pass integers by const reference?

A: Well, we shouldn’t.

It is much more important when we are talking about structs and classes.

Temporary objects and references

```
struct Point {int x; int y};

void foo(Point& p) {
   cout << p.x << x.y;
}

Point x{13, 42};
...
foo(x); // ok
```

Questions:

Q: Why the hell should we pass integers by const reference?

A: Well, we shouldn’t.

It is much more important when we are talking about structs and classes.

Temporary objects and references

```
struct Point {int x; int y};

void foo(Point& p) {
   cout << p.x << x.y;
}

Point x{13, 42};
...
foo(x); // ok
foo(Point{0, 0}); // error
```

temporary obj

Questions:

Q: Why the hell should we pass integers by const reference?

A: Well, we shouldn't.

It is much more important when we are talking about structs and classes.

Temporary objects and references

```
struct Point {int x; int y};

void foo(const Point& p) {
   cout << p.x << x.y;
}

Point x{13, 42};
...
foo(x); // ok
foo(Point{0, 0}); // ok
```

Questions:

Q: Why the hell should we pass integers by const reference?

A: Well, we shouldn’t.

It is much more important when we are talking about structs and classes.

Temporary objects and references

```
struct Point {int x; int y};

void foo(const Point& p) {
    cout << p.x << x.y;
}

Point x{13, 42};
...
foo(x); // ok
foo(Point{0, 0}); // ok
```

Questions:

Q: Why the hell should we pass integers by const reference?

A: Well, we shouldn’t.

It is much more important when we are talking about structs and classes.

Why?

Temporary objects and references

```
struct Point {int x; int y};

void foo(const Point& p) {
   cout << p.x << x.y;
}

Point x{13, 42};
...
foo(x); // ok
foo(Point{0, 0}); // ok
```

Questions:

Q: Why the hell should we pass integers by const reference?

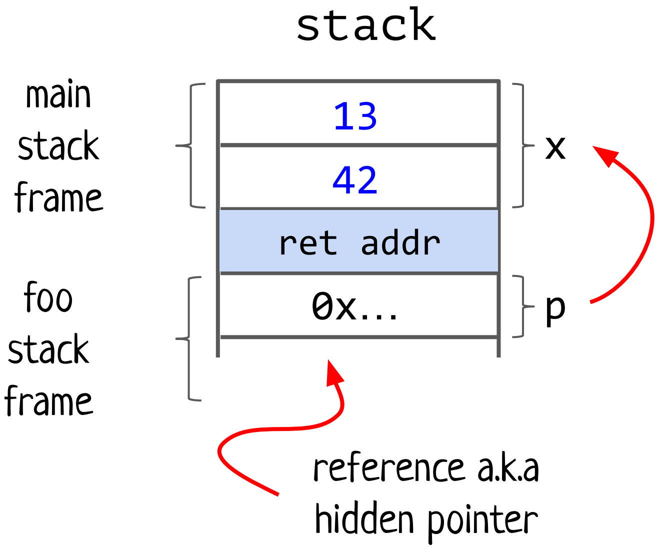
A: Well, we shouldn’t.

It is much more important when we are talking about structs and classes.

Why? To avoid copying!

Temporary objects and references

```
struct Point {int x; int y};

void foo(const Point& p) {
    cout << p.x << x.y;
}

Point x{13, 42};
...
foo(x); // ok
foo(Point{0, 0}); // ok
```



Dangling things

The Lion King © Disney 1994 / mylionking.com

Dangling pointers

```
void foo() {
    Vector* pv = nullptr;
    {
        Vector v{0};
        pv = &v;
    }
    pv->push_back(10);
    std::cout << pv->pop_back();
}
```

Dangling pointers

```
void foo() {
    Vector* pv = nullptr;
    {
        Vector v{0};
        pv = &v;
    }
    pv->push_back(10);
    std::cout << pv->pop_back();
}
```

lifetime of v

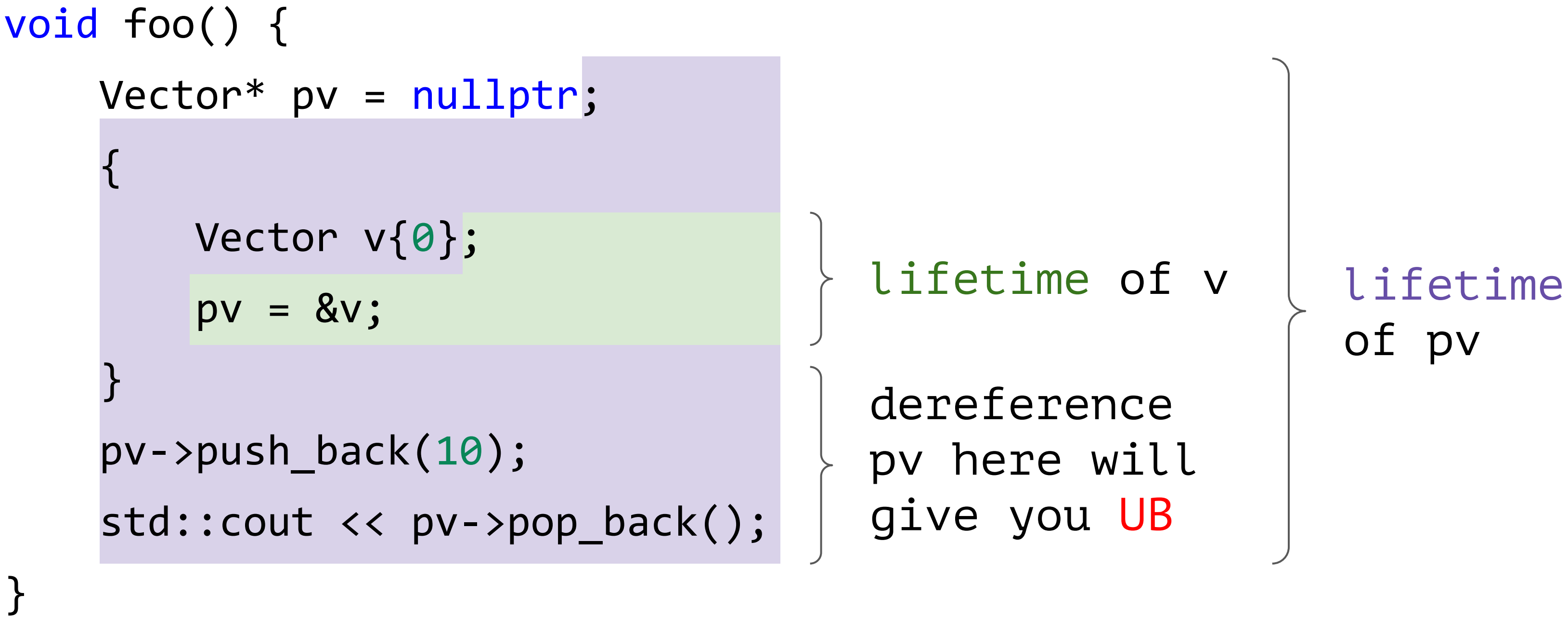
Dangling pointers

```
void foo() {
    Vector* pv = nullptr;
    {
        Vector v{0};
        pv = &v;
    }
    pv->push_back(10);
    std::cout << pv->pop_back();
}
```

lifetime of v

lifetime of pv

Dangling pointers



```
void foo() {
    Vector* pv = nullptr;
    {
        Vector v{0};
        pv = &v;
    }
    pv->push_back(10);
    std::cout << pv->pop_back();
}
```

lifetime of v

dereference pv here will give you UB

lifetime of pv

Dangling pointers

```
void foo() {
    Vector* pv = nullptr;
    {
        Vector v{0};
        pv = &v;
    }
    pv->push_back(10);
    std::cout << pv->pop_back();
}
```

lifetime of v

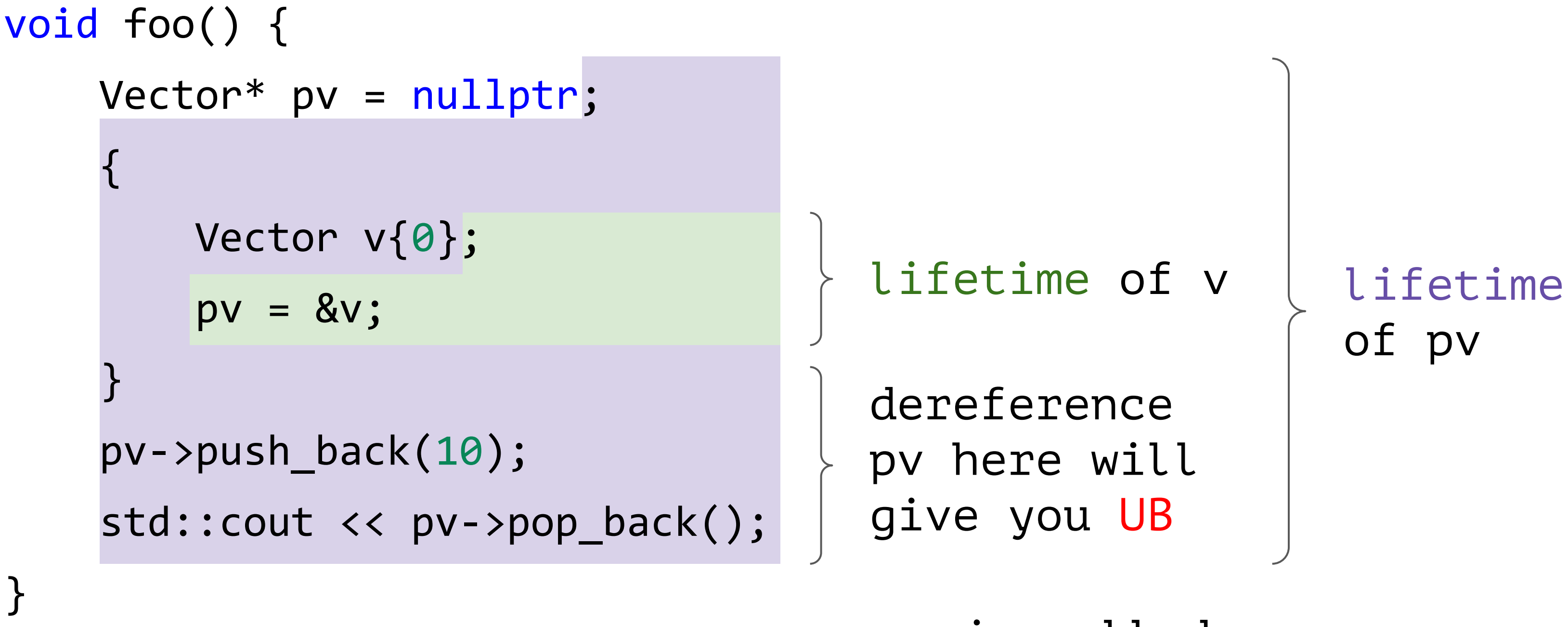
dereference pv here will give you UB

lifetime of pv

pv is called dangling pointer here

Dangling pointers

Is the same possible with references?



```
void foo() {
    Vector* pv = nullptr;
    {
        Vector v{0};
        pv = &v;
    }
    pv->push_back(10);
    std::cout << pv->pop_back();
}
```

lifetime of v

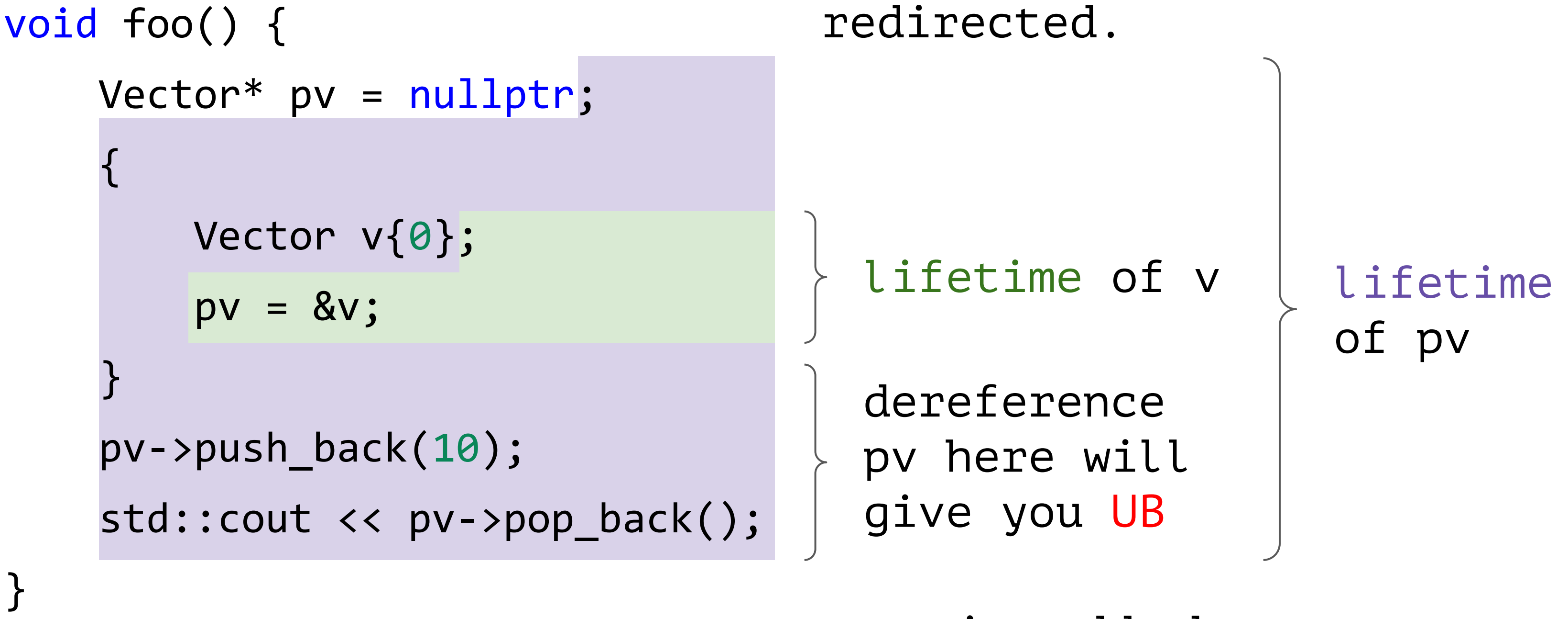
dereference pv here will give you UB

lifetime of pv

pv is called dangling pointer here

Dangling pointers

Is the same possible with references? Well, this concrete example it is not, as references can’t be redirected.



```
void foo() {
    Vector* pv = nullptr;
    {
        Vector v{0};
        pv = &v;
    }
    pv->push_back(10);
    std::cout << pv->pop_back();
}
```

pv is called dangling pointer here

Dangling pointers

Is the same possible with references? Well, it this concrete example is not, as references can’t be redirected, but…

```
void foo() {
    Vector* pv = nullptr;
    {
        Vector v{0};
        pv = &v;
    }
    pv->push_back(10);
    std::cout << pv->pop_back();
}
```

lifetime of v

dereference pv here will give you UB

lifetime of pv

pv is called dangling pointer here

Dangling references

```
Vector& terrible() {
    Vector local{16};
    return local;
}

int main() {
    Vector& rv = terrible();
    rv.push_back(13);
    std::cout << rv.pop_back();
    return 0;
}
```

Dangling references

```
Vector& terrible() {
    Vector local{16};
    return local;
}

int main() {
    Vector& rv = terrible();
    rv.push_back(13);
    std::cout << rv.pop_back();
    return 0;
}
```

lifetime of local

Dangling references

```
Vector& terrible() {
    Vector local{16};
    return local;
}
```

lifetime of local

```
int main() {
    Vector& rv = terrible();
    rv.push_back(13);
    std::cout << rv.pop_back();
    return 0;
}
```

lifetime* of rv

Dangling references

```
Vector& terrible() {
    Vector local{16};
    return local;
}
```

lifetime of local

```
int main() {
    Vector& rv = terrible();
    rv.push_back(13);
    std::cout << rv.pop_back();
    return 0;
}
```

lifetime* of rv

*formally saying rv is not an object, but spec describes its lifetime as if it is a scalar

Dangling references

```
Vector& terrible() {
    Vector local{16};
    return local;
}
```

lifetime of local

```
int main() {
    Vector& rv = terrible();
    rv.push_back(13);
    std::cout << rv.pop_back();
    return 0;
}
```

deref here is UB

lifetime* of rv

Dangling references

```
int main() {
    Vector* vs = new Vector[16];
    Vector& rv5 = vs[5];
    delete[] vs;
    std::cout << rv5.pop_back();
    return 0;
}
```

Just another example of dangling reference (after deallocation)

Dangling references

```
int main() {
    Vector* vs = new Vector[16];
    delete[] vs;
    Vector& rv5 = vs[5];
    std::cout << rv5.pop_back();
    return 0;
}
```

Just another example of dangling reference (from the very beginning)

Dangling references

And what about temporary objects? They should be an endless source of dangling reference!

```
class Vector {
   Vector(int capacity) ... {
       cout << "constructor called" << endl;
   }
   ~Vector() {
       ...
       cout << "destructor called" << endl;
   }
};

int main() {
    Vector{16};
    cout << "after expression" << endl;
    return 0;
}
```

Object lifetime (second approximation)

When object dies?

If object is temporary => end of the full statement;

Otherwise, depends on its storage duration:

- static => when program terminates (return from main or call exit)
- automatic => at the end of the scope
- dynamic => when delete is called

```
class Vector {
   Vector(int capacity) ... {
       cout << "constructor called" << endl;
   }
   ~Vector() {
       ...
       cout << "destructor called" << endl;
   }
};

int main() {
    Vector{16};
    cout << "after expression" << endl;
    return 0;
}
```

What will be printed?

```
class Vector {
   Vector(int capacity) ... {
       cout << "constructor called" << endl;
   }
   ~Vector() {
       ...
       cout << "destructor called" << endl;
   }
};
```

```
int main() {
    Vector{16};
    cout << "after expression" << endl;
    return 0;
}
```

Output:

```
    constructor called
    destructor called
    after expression
```

```
class Vector {
   Vector(int capacity) ... {
       cout << "constructor called" << endl;
   }
   ~Vector() {
       ...
       cout << "destructor called" << endl;
   }
};
```

```
int main() {
    Vector{16};
    cout << "after expression" << endl;
    return 0;
}
```

lifetime of temp object

Output:

```
    constructor called
    destructor called
    after expression
```

Dangling references

And what about temporary objects? They should be an endless source of dangling reference!

```
int main() {
    const Vector& rv = Vector{16};
    cout << "after expression" << endl;
    cout << rv.capacity() << endl;
    return 0;
}
```

Dangling references

And what about temporary objects? They should be an endless source of dangling reference!

temporary object

```
int main() {
    const Vector& rv = Vector{16};
    cout << "after expression" << endl;
    cout << rv.capacity() << endl;
    return 0;
}
```

Is rv dangling reference?

Dangling references

And what about temporary objects? They should be an endless source of dangling reference!

```cpp
int main() {
    const Vector& rv = Vector{16};
    cout << "after expression" << endl;
    cout << rv.capacity() << endl;
    return 0;
}
```

Output:

```
    constructor called
    after expression
    16
    destructor called
```

Dangling references

And what about temporary objects? They should be an endless source of dangling reference!

temporary object

```
int main() {
    const Vector& rv = Vector{16};
    cout << "after expression" << endl;
    cout << rv.capacity() << endl;
    return 0;
}
```

Is rv dangling reference?

Actually no! Const lvalue reference prolongs lifetime of temporary objects to its lifetime!

Dangling references

And what about temporary objects? They should be an endless source of dangling reference!

```
int main() {
    const Vector& rv = Vector{16};
    cout << "after expression" << endl;
    cout << rv.capacity() << endl;
    return 0;
}
```

(prolonged) lifetime of temporary object

Object lifetime (second approximation)

When object dies?

If object is temporary => end of the full statement;

Otherwise, depends on its storage duration:

- static => when program terminates (return from main or call exit)
- automatic => at the end of the scope
- dynamic => when delete is called

Object lifetime (third approximation)

When object dies?

If object is temporary =>

- if it is bound to some reference => lifetime extended to this reference;

- otherwise, end of the full statement;

Otherwise, depends on its storage duration:

- static   => when program terminates
- automatic => at the end of the scope
- dynamic   => when delete is called

Dangling references

And what about temporary objects? They should be an endless source of dangling reference!

```
int main() {
    const Vector& rv = Vector{16};
    cout << "after expression" << endl;
    cout << rv.capacity() << endl;
    return 0;
}
```

(prolonged) lifetime of temporary object

Dangling references

And what about temporary objects? They should be an endless source of dangling reference!

But how far can it go?

```
int main() {
    const Vector& rv = Vector{16};
    cout << "after expression" << endl;
    cout << rv.capacity() << endl;
    return 0;
}
```

(prolonged) lifetime of temporary object

Dangling references

But how far can it go? What if reference (its scalar) is placed in dynamic memory? Will local temporary object be alive, well, until delete is called?

```
int main() {
    const Vector& rv = Vector{16};
    cout << "after expression" << endl;
    cout << rv.capacity() << endl;
    return 0;
}
```

(prolonged) lifetime of temporary object

Dangling references

```
struct Container {
    const Vector& ref;
};

int main() {
    Container c = Container{Vector{16}};
    cout << "after expression" << endl;
    cout << c.ref.capacity() << endl;
    return 0;
}
```

Dangling references

```
struct Container {
    const Vector& ref;
};

int main() {
    Container c = Container{Vector{16}};
    cout << "after expression" << endl;
    cout << c.ref.capacity() << endl;
    return 0;
}
```

```
Output:
    constructor is called
    after expression
    16
    destructor is called
```

Dangling references

```
struct Container {
    const Vector& ref;
};

int main() {
    Container c = Container{Vector{16}};
    cout << "after expression" << endl;
    cout << c.ref.capacity() << endl;
    return 0;
}
```

Well, looks like everything works 😊

Output:

```
    constructor is called
    after expression
    16
    destructor is called
```

Dangling references

```
struct Container {
    const Vector& ref;
};

int main() {
    Container* c = new Container{Vector{16}};
    cout << "after expression" << endl;
    cout << c->ref.capacity() << endl;
    delete c;
    return 0;
}
```

```
Output: ???
```

Dangling references

```
struct Container {
    const Vector& ref;
};

int main() {
    Container* c = new Container{Vector{16}};
    cout << "after expression" << endl;
    cout << c->ref.capacity() << endl;
    delete c;
    return 0;
}
```

```
Output:
    constructor is called
```

Dangling references

```
struct Container {
    const Vector& ref;
};

int main() {
    Container* c = new Container{Vector{16}};
    cout << "after expression" << endl;
    cout << c->ref.capacity() << endl;
    delete c;
    return 0;
}
```

Wait, what?

```
Output:
    constructor is called
    destructor is called
```

Dangling references

```
struct Container {
    const Vector& ref;
};

int main() {
    Container* c = new Container{Vector{16}};
    cout << "after expression" << endl;
    cout << c->ref.capacity() << endl;
    delete c;
    return 0;
}
```

Wait, what?

Output:

```
    constructor is called
    destructor is called
    after expression
```

Dangling references

```
struct Container {
    const Vector& ref;
};

int main() {
    Container* c = new Container{Vector{16}};
    cout << "after expression" << endl;
    cout << c->ref.capacity() << endl;
    delete c;
    return 0;
}
```

Wait, what?

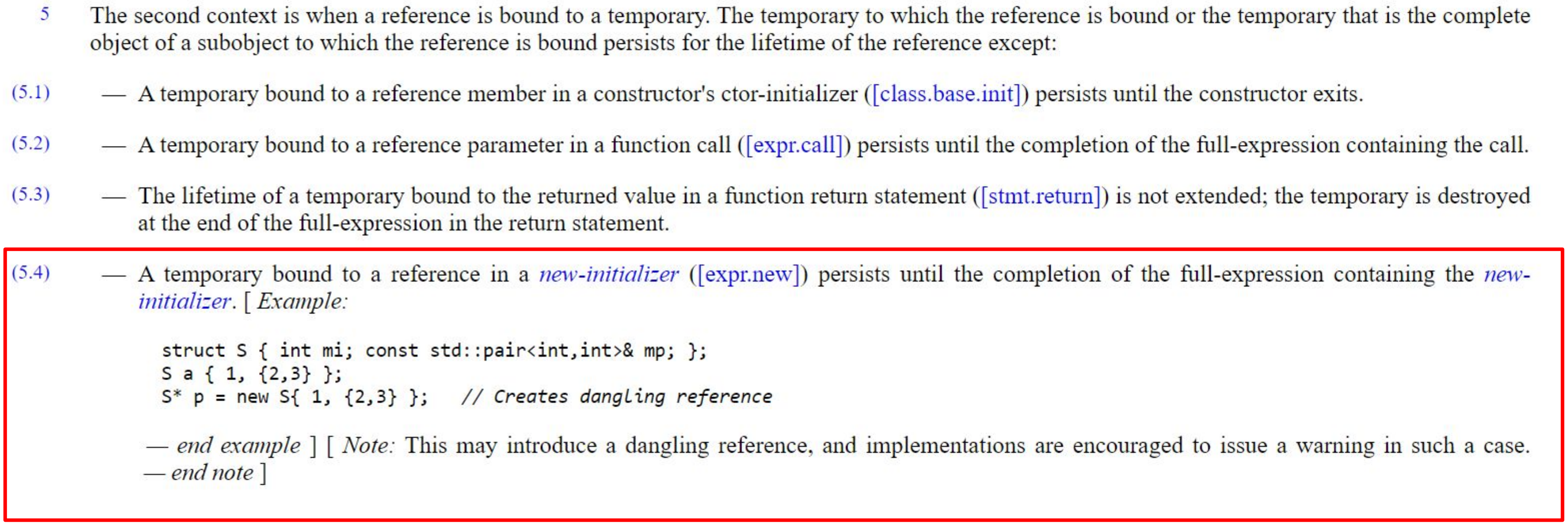
```
Output:
    constructor is called
    destructor is called
    after expression
    undefined
```

Dangling references

https://timsong-cpp.github.io/cppwp/n3337/class.temporary

Dangling references

https://timsong-cpp.github.io/cppwp/n3337/class.temporary



5 The second context is when a reference is bound to a temporary. The temporary to which the reference is bound or the temporary that is the complete object of a subobject to which the reference is bound persists for the lifetime of the reference except:

(5.1) — A temporary bound to a reference member in a constructor's ctor-initializer ([class.base.init]) persists until the constructor exits.

(5.2) — A temporary bound to a reference parameter in a function call ([expr.call]) persists until the completion of the full-expression containing the call.

(5.3) — The lifetime of a temporary bound to the returned value in a function return statement ([stmt.return]) is not extended; the temporary is destroyed at the end of the full-expression in the return statement.

(5.4) — A temporary bound to a reference in a *new-initializer* ([expr.new]) persists until the completion of the full-expression containing the *new-initializer*. [*Example:*

```
struct S { int mi; const std::pair<int,int>& mp; };
S a { 1, {2,3} };
S* p = new S{ 1, {2,3} };   // Creates dangling reference
```

— *end example*] [*Note:* This may introduce a dangling reference, and implementations are encouraged to issue a warning in such a case. — *end note*]

Dangling references

Temporary objects lifetime prolongation with const lvalue references is very fragile. You should absolutely understand what are you doing and know the specification.

Dangling references

Temporary objects lifetime prolongation with const lvalue references is very fragile. You should absolutely understand what are you doing and know the specification.

Or avoid just it!

Dangling references

Temporary objects lifetime prolongation with const lvalue references is very fragile. You should absolutely understand what are you doing and know the specification.

Or avoid just it!

Storing const lvalue refs in fields is just a minefield.

Think twice whether to use it if you still want to have two legs.

Takeaways

- References as encapsulated pointers (but no silver bullets)

Takeaways

- References as encapsulated pointers (but no silver bullets)
- const modifier for:
  ✓ compile time checks,
  ✓ work with temporary objects.

Takeaways

- ○ References as encapsulated pointers (but no silver bullets)
- ○ const modifier for:
  - ✓ compile time checks,
  - ✓ work with temporary objects.
- ○ First meet with temporary objects
  - ✓ Their purpose
  - ✓ Their lifetime (and its prolongation with const lvalue refs)

Not So Tiny Task №2 (1 points)

Prepare a small framework to work with lines on a plane.

- Define a struct/class for Point and Line.
- Add constructors from 2 points and from coefficients.
- Add functions to find intersection with other line, finding a perpendicular line at some point.

Your solution should avoid copying structures and try be as safe as possible (via const). Don't forget about tests and sanitizers.